Computing Systems - Reading List

Module Name: Shell Scripting
Lesson 1: Shell Scripting
Recommended Reading Material
Brief: Notes attached with this document
Book:
Remote access link (Optional):
Duration: 60 minutes

Objectives

1. Introduction to Unix Shell and Pattern Matching
2. Shell Script
3. Interactive Shell Scripts
4. Performing Arithmetic Operations
5. Variables in Shell
6. Decision making constructs in Shell
7. More Decision-Making Constructs in Shell
8. Looping Constructs in Shell
9. Arrays

1. What is Shell?

A program that interprets commands and allows a user to execute commands by typing them manually at a terminal, or automatically in programs called ***shell scripts*.**
A shell is *not* an operating system. It is a way to interface with the operating system and execute commands.
Login shell is BASH = **B**ourne **A**gain **Sh**ell
Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems. It has all the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line. Since it is Free Software, it has been adopted as the default shell on most Linux systems.
Many commands accept arguments which are file names. For example:

ls -l main.c

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern.
For example,

ls -l *.c

generates, as arguments to *ls,* all file names in the current directory that end in ***.c***.
The character * is a pattern that will match any string including the null string. This mechanism is useful both to save typing and to select names according to some pattern.
Following are the special characters interpreted by the shell called as ***Wild cards.***

Command	What does the command do?
*	Matches any number of characters including none
?	Matches a single character

[ijk]	Matches a single character either i , j or k
[!ijk]	Not i, j or k
[x-z]	At least a single character within this ASCII range
[!x-z]	Not a single character within this range

(i) * stands for zero or more characters.

Examples:

ls a **(output is only a if a is a file, output is the content of a if a is a directory)**

ls ar* **(all those that start with ar)**

ls *vict **(all those files that end with vict)**

ls *[ijk] **(all files having i ,j or k as the last character of the filename)**

ls *[b-r] **(all files which have at least any character between b and r as the last character of the filename)**

ls * **(all files)**

(ii) ? matches a single character

Examples:

ls a?t matches all those files of three characters with **a** as the first and **t** as the third character and any character in between

ls ?oo matches all three character files whose filename end with **oo**

What will the following commands give?

Command	What does it do?
ls ??i*	Matches any number of characters but definitely two characters before i followed by any number of characters
ls ?	Matches all the single character files
ls *?	Matches any file with at least 1 character
ls ?*	Matches any file with at least 1 character
ls "*"	Matches any file with * as the filename (exactly 1 character which is *)
ls "*"*	Matches all files starting with * as filename

1.1 Quoting

Characters that have a special meaning to the shell, such as **< > * ? | &**, are called *metacharacters*. Any character preceded by a \ is *quoted* and loses its special meaning, if any.

The \ is elided so that

echo \?

will echo a single **?** and

**echo **

will echo a single \

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes.

For example,

echo xx'**'xx**

will echo

xx**xx**

The quoted string may not contain a single quote but may contain new lines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters.

2. Shell Scripts

The shell may be used to read and execute commands contained in a file.

As a simple case, assume that we have a scenario where commands who, ls and date are executed one after the other.

To write shell scripts, open a new file in gedit (say gedit firstShellPgm)

Type all the commands that you want to get executed one after the other. i.e. who, ls and date.

The file content will be now

```
echo “My First Shell Program”
who
ls -l
date
```

As stated earlier, the shell may be used to read and execute commands contained in a file.

To read and execute, the file use the following in command line

sh file [args ...] ***e.g. sh firstShellPgm***

sh calls the shell to read commands from *file.* Such a file is called a *command procedure* or *shell procedure or shell script.*

Execute the program by typing

sh ***firstShellPgm***

This produces the output as shown in Figure 1.

Note: You need to give execute permission to file firstShellPgm

chmod +x firstShellPgm

```
user@Z-101:~$ gedit firstShellPgm
user@Z-101:~$ sh firstShellPgm
"My First Shell Program"
user     tty7         2018-01-16 14:18 (:0)
total 60
drwxr-xr-x 2 user user 4096 Nov 15 10:33 Desktop
drwxr-xr-x 2 user user 4096 Nov  5 10:52 Documents
drwxr-xr-x 3 user user 4096 Nov 28 16:19 Downloads
-rw-rw-r-- 1 user user   49 Jan 16 14:49 firstShellPgm
drwxrwxr-x 3 user user 4096 Nov 28 17:12 java
drwxrwxr-x 4 user user 4096 Jan 13 09:44 Lab1
drwxrwxr-x 4 user user 4096 Nov 28 17:13 labexam
drwxr-xr-x 2 user user 4096 Nov  5 10:52 Music
drwxrwxr-x 4 user user 4096 Nov 14 19:12 onlinetest
drwxr-xr-x 2 user user 4096 Jun 15  2017 Pictures
drwxr-xr-x 2 user user 4096 Jun 15  2017 Public
drwxr-xr-x 2 user user 4096 Jun 15  2017 Templates
drwxrwxr-x 4 user user 4096 Jan 13 10:05 test
drwxr-xr-x 2 user user 4096 Jun 15  2017 Videos
drwxrwxr-x 5 user user 4096 Nov 28 15:35 workspace
Tue Jan 16 14:50:09 IST 2018
user@Z-101:~$
```

Figure 1: Output of the sh firstShellPgm

Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...**

For example, if the file ***secondShellPgm*** contains

```
echo My Second Shell Program with Parameter Passing
echo The parameter1 is $1 and Parameter 2 is $2
who|grep $1*
ls -l $2*
date
```

then

sh *secondShellPgm* CPSec first

produces the result as shown in Figure 2.

```
CPSec01@Z-188:~$ gedit secondShellPgm
CPSec01@Z-188:~$ sh secondShellPgm CPSec first
My Second Shell Program with Parameter Passing
The parameter1 is CPSec and Parameter 2 is first
CPSec01  tty7         2018-01-18 01:45 (:0)
-rw-rw-r-- 1 CPSec01 CPSec01 3 Jan 17 20:21 firstShellPgm
Wed Jan 17 20:21:17 IST 2018
CPSec01@Z-188:~$
```

Figure 2: Output of the sh secondShellPgm CPSec first

When the file executes, the first argument (CPSec in the previous example) replaces $1, second argument replaces $2 …….

So

echo The parameter1 is $1 and Parameter 2 is $2

is equivalent to

echo The parameter1 is CPSec and Parameter 2 is first

and

who|grep $1*

is equivalent to

who|grep CPSec*

and

ls -l $2*

is equivalent to

ls -l first*

i.e. CPSec is assigned to variable 1 and first is assigned to variable 2

Note: You need to give execute permission to file secondShellPgm

chmod +x secondShellPgm

3. Interactive Shell Scripts

Interaction with the computer is always through input and output operations. The ***read*** command allows the shell scripts to read input from the user. Let us write our first interactive shell script here. You may save it in the file ***first***.

The script is as follows

```
#this script asks user to input his name and then prints his name
echo What is your name \?
read name
echo Hello $name. Happy Programming.
```

Execute the script by typing

sh first

You must have noticed \ symbol before? This can be avoided if the message to be displayed is enclosed in quotes. This script causes the name entered by you is stored in the variable ***name***. Also, while using in echo statement, this variable name is attached with $ indicating that name is a variable, and extract the value stored in the variable ***name*** and use it in place of it.

The output is:

```
What is your name ?
john
Hello john. Happy Programming.
```

4. Performing Arithmetic operations

We start with an example here. Type the following code in the file named ***arithmetic***.

```
#this script performs different arithmetic operations on two numbers
a=$1
b=$2
echo `expr $a + $b`
echo `expr $a - $b`
echo `expr $a \* $b`        #multiplication
echo `expr $a / $b`
echo `expr $a % $b`         #modular division, returns reminder
```

Now execute the script by typing **sh arithmetic 20 12**

You will get answer as follows

```
32
8
240
1
8
```

Line 1 is commented which is used for improving readability of the script. This line will not be executed. Anything following # is a comment and will be only for improving readability.

Line 2 and 3 assigns some values [passed through argument 1 and argument 2] to variables a and b. In the line echo `expr $a + $b` expr is the key value that signifies it as an arithmetic expression to be evaluated.

An expression like **$a * \($b + $c \) / $d** is a valid expression. This expression performs **a * (b+c) / d**. The enclosing punctuation (` `) in **`expr $a + $b`** is accent grave. It causes the expression to be evaluated. expr is only able to carry out the integer arithmetic.

NOTE: The symbol quoting expr can be found on key having Tilde (~) below ESC key.

5. Variables in Shell

To process data, it must be kept in computer's RAM. RAM is divided into small locations, and each location had unique number called memory location/address, which is used to hold the data. Programmer can give a unique name to this memory location/address called memory variable or variable [It is a named storage location that may take different values, but only one at a time]. In Linux (Shell), there are two types of variable:

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

(2) **User defined variables (UDV)** - Created and maintained by user.

5.1 System Variables

Some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Shell name
BASH_VERSION=4.3.48(1)-release	Shell version name
COLUMNS=80	Number of columns for the screen
HOME=/home/CPSec01	Home directory
LINES=24	Number of columns for the screen

LOGNAME=CPSec01	Log in name
OSTYPE=linux-gnu	OS type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Path settings
PS1=[\u@\h \W]\$	Prompt settings
PWD=/home/CPSec01/	Present working directory
SHELL=/bin/bash	Shell name
USERNAME=CPSec01	User name who is currently login to this PC

Caution: Do not modify System variables. This can some time create problems.
You can see the current values of these variables by typing **echo $variable name**
Example: echo $HOME

5.2 User Defined Variable (UDV)

Defining a User defined Variable

To define a UDV, the following syntax is used.

Variable name=value

'**value**' is assigned to given '**variable name**' and Value must be on right side of the = sign.

Examples:

```
number=10  # this is ok

10=number  # Error, Value must be on right side of the = sign.

vech=Bus   # defines variable called 'vech' having value Bus
```

Rules for Naming variable name (Both UDV and System Variable)

1. Variable name must begin with Alphanumeric character or underscore character, followed by one or more Alphanumeric character.
   Examples of valid shell variable are:
   HOME
   SYSTEM_VERSION
   vech
   number
2. Don't put spaces on either side of the equal sign when assigning value to variable.
   For example, in following variable declaration there will be no error
   number=10
   But there will be problem for any of the following variable declaration:
   number =10
   number= 10
   number = 10
3. Variables are case-sensitive, just like filenames in Linux.
   For example,
   number=10
   Number=11

NUMBER=20

Above all are different variable names.

To print value 20 we have to use **echo $NUMBER**.

echo $number # will print 10 but not 20

echo $Number # will print 11 but not 20

4. You can define NULL variable as follows [NULL variable is variable which has no value at the time of definition]
   For example
   vech=
   vech=""
   Try to print it's value by issuing the command **echo $vech**
   Nothing will be shown because variable has no value i.e. NULL variable.
5. Do not use **?,*** etc, to name your variable names.

6. Decision making constructs in Shell

What computer know is 0 (zero) and 1 that is Yes or No.

To make this idea clear, lets take the help of bc - Linux calculator program. Type

bc

After this command bc is started and waiting for your commands, i.e. give it some calculation as follows type 5 + 2 as: **5 + 2**

7

7 is response of bc i.e. addition of 5 + 2.

Now see what happened if you type 5 > 2

5 > 2

1

1 is response of bc, How? bc compare 5 with 2 as, Is 5 is greater than 2, bc gives the answer as 'YES' by showing 1 as value. Now try

5 < 2

0

0 indicates FALSE. i.e., Is 5 is less than 2?, the answer NO indicated by bc is by showing 0.

Remember in bc, logical operations always returns **true** (1) or **false** (0).

Try following in bc to clear your Idea and not down bc's response.

5 > 12

5 == 10

5 != 2

5 == 5

1= < 2

Expression	Meaning to us	Your Answer	BC's Response
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1

1 < 2	Is 1 is less than 2	YES	1

It means whenever there is any type of comparison in Linux Shell, it gives only one of the two answers YES or NO.

In Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value	No/False	-1, 32, 55 anything but not zero

Remember bc and Linux Shell uses two *different ways to show True/False values*

Value	Shown in bc as	Shown in Linux Shell as
True/Yes	1	0
False/No	0	Non - zero value

6.1. if condition

if condition is used for decision making in shell script. If the given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1
        ...
fi
```

Condition is defined as:

"*Condition is nothing but comparison between two values.*"

For comparison you can use test or [expression] statements.

Expression is defined as:

"An expression is nothing but combination of values, relational operator (such as >,<, <> etc) and mathematical operators (such as +, -, / etc)."

Following are all examples of expression:

```
5 > 2
3 + 6
3 * 65
a < b
c > 5 + 30 -1
```

test command or [expression]

test command or [expression] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax: **test expression OR [expression]**

Example: Following script determine whether given argument number is positive.

```
if test $1 -gt 0# if [ $1 –gt 0 ] will also work
then
echo "$1 number is positive"
fi
```

Run it as follows
chmod +x ispostive
sh ispostive 5
5 number is positive
sh ispostive -45
Nothing is printed
sh ispostive
Test and justify the output
Detailed explanation
The line if test $1 -gt 0, test to see if first command line argument ($1) is greater than 0. If it is true(0) then test will return 0 and output will printed as 5 number is positive but for -45 argument there is no output because our condition is not true (no -45 is not greater than 0) hence echo statement is skipped.
For last statement we have not supplied any argument hence error **ispostive: 1: test: -gt: unexpected operator**, is generated by shell , to avoid such error we can test whether command line argument is supplied or not.
test or [expression] works with
1.Integer (Number without decimal point)
2.File types
3.Character strings

For Mathematics, use following operator in Shell Script

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expression] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

NOTE: == is equal, != is not equal.

For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist

-z string1	string1 is NULL and does exist

Shell also test for file and directory types

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators

Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

6.2 if...else...fi

If given condition is true then command1 is executed otherwise command2 is executed.

Syntax:

```
if condition
 then
        condition is zero (true - 0)
        execute all commands up to else statement
 else
        if condition is not true then
        execute all commands up to fi
 fi
```

For e.g. Write Script in file isPos as follows:

```
 if [ $# -eq 0 ]   # $# is used to see how many parameters passed
 then
   echo "You must supply one integer"
   exit 1                    # exit from the shell script
 fi
 if test $1 -gt 0
 then
   echo "$1 number is positive"
 else
   echo "$1 number is negative"
 fi
```

Do the following:

chmod +x isPos

sh isPos 5

5 number is positive

sh isPos -45
-45 number is negative
sh isPos
You must supply one integers
sh isPos 0
0 number is negative

Detailed explanation
First script checks whether command line argument is given or not, if not given then it print error message as "*You must supply one integers*". if statement checks whether number of argument ($#) passed to script is not equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true. And finally statement exit 1 causes normal program termination with exit status 1.
The last sample run **sh isPos 0** , gives output as "*0 number is negative*", because given argument is not > 0, hence condition is false and it's taken as negative number. To avoid this replace second if statement with **if test $1 -ge 0**.

7. More Decision Making Constructs in Shell

7.1. Nested if-else-fi

You can write the entire if-else construct within either the body of if statement of the body of else statement. This is called the nesting of ifs. Consider a file named **isPos_nestedif** with the following content.

```
if [ $# -eq 0 ]
then
     echo "You must supply one integer"
else

     if test $1 -ge 0
     then
         echo "$1 number is positive"
     else
         echo "$1 number is negative"
     fi

fi
```

Run the above shell script as follows:

```
chmod +x isPos_nestedif
sh isPosnestedif 20

20 number is positive
```

Note that Second *if-else* construct is nested in the first *else* statement. If condition in the first '*if*' statement is false the condition in the second '*if*' statement is checked. If it is false as well the final *else* statement is executed.

You can use the nested *if*s as follows also:
Syntax:

```
if condition
then
     if condition
     then
          do this
     else
          do this
     fi
else

          do this
fi
```

7.2. Multilevel if-then-else

Syntax:

```
if condition
then
    condition is zero (true - 0)
    execute all commands up to elif statement
elif condition1
then
    condition1 is zero (true - 0)
    execute all commands up to elif statement
else
    None of the above condition, condtion1, condtion2
    are true (i.e. all of the above nonzero or false)
    execute all commands up to fi
fi
```

For multilevel if-then-else statement try the following script in a file named **isPos_Multilevelif**:

```
if [ $# -eq 0 ]
then
     echo "You must supply one integer"
elif test $1 -gt 0
then
    echo "$1 number is positive"
elif test $1 -lt 0
```

```
then
    echo "$1 number is negative"
elif test $1 -eq 0
then
    echo "$1 number is zero"
else
    echo "Opps! $1 is Not a Number, supply a Number"
fi
```

Try above script as follows:

chmod +x isPos_Multilevelif

sh isPos_Multilevelif 20

sh isPos_Multilevelif 0

sh isPos_Multilevelif -20

sh isPos_Multilevelif CP

7.3. The case Statement

The case statement is a good alternative to multilevel if-then-else-fi statement. It enables you to match several values against one variable. It's easier to read and write.

Syntax:

```
case  $variable-name  in
    pattern1)     command
                  command;;
    pattern2)     command
                  command;;
    patternN)     command
                  command;;
    *)            command
                  command;;
esac
```

The *$variable-name* is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and it's executed if no match is found. For example write script named **vehicle** as follows:

```
rental=$1
case $rental in
"car") echo "For $rental Rs.400 per day";;
"van") echo "For $rental Rs.500 per day";;
"jeep") echo "For $rental Rs.300 per day";;
```

```
"bicycle") echo "For $rental Rs.20 per day";;
*) echo "Sorry, I cannot get a $rental for you";;
esac
```

Save it and run it as follows:

chmod +x vehicle
sh vehicle van
sh vehicle car
sh vehicle Maruti-800

Note that esac is always required to indicate end of case statement.

8. Looping Constructs in Shell

Loop is defined as:

"A Program *can repeat particular set of instructions again and again, until particular condition satisfies. A group of instructions that is executed repeatedly is called a loop.*"

Bash supports:

- for loop
- while loop

Note that in each and every loop,

- First, the variable used in loop condition must be initialized, then execution of the loop begins.
- A test (condition) is made at the beginning of each iteration.
- The body of loop ends with a statement that modifies the value of the test (condition) variable.

8.1. for Loop

Syntax **:**

```
for { variable name } in { list }
do
     execute one for each item in the list
     until the list is finished (And repeat all
     statement between do and done)
done
```

For example write script named **display_num** as follows:

```
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

Run the above script as follows:

```
chmod +x display_num
sh display_num
```

The for loop first creates i variable and assignes a number to i from the list of numbers [from 1 to 5]. The shell executes echo statement for each assignment of i. (This is usually know as iteration). This process will continue until all the items in the list are done. So it will repeat 5 echo statements.

You can use following syntax as well:

Syntax:

```
for (( expr1; expr2; expr3 ))
do
     repeat all statements between do and
     done until expr2 is TRUE
done
```

In the above syntax, before the first iteration, ***expr1*** is evaluated. This is usually used to initialize variables for the loop.

All the statements between do and done is executed repeatedly UNTIL the value of ***expr2*** is TRUE.

After each iteration of the loop, ***expr3*** is evaluated. This is usually used to increment a loop counter.

For example write script named **display_num1** as follows:

```
for (( i=0 ; i<=5; i++ ))
do
   echo "Welcome $i times"
done
```

Run the above script as follows:

```
chmod +x display_num1
bash display_num1
```

In the above example, first expression (i=0) is used to set the variable **i**'s value to zero. Second expression is a condition i.e. all statements between do and done executed as long as expression 2 is TRUE. (i.e. continue the execution as long as the value of variable **i** is less

than or equal to 5). The last expression, **i++** increments the value of **i** by 1 i.e. it's equivalent to i=i+1 statement.

8.2 Nesting of for Loop

Loop statement can also be nested similar to the if statements. You can nest the for loop. To understand the nesting of for loop, see the following shell script named **display_num_nested**.

```
for (( i=1; i<=5; i++ ))          # Outer
for loop
do
    for (( j=1; j<=5; j++ ))     # Inner for loop
    do
          echo  "$i,$j   "
    done
  echo ""                         #print the new line
done
```

Run the above script as follows:
chmod +x display_num_nested
bash display_num_nested

Here, for each value of **i,** the inner loop is cycled through 5 times, with the variable **j** taking values from 1 to 5. The inner for loop terminates when the value of **j** exceeds 5, and the outer loop terminates when the value of **i** exceeds 5.

Infinite loop

Infinite for loop can be created with empty expressions, such as

```
for (( ; ; ))
do
   echo "infinite loops [ hit CTRL+C to stop]"
done
```

Conditional exit with break

You can do early exit from a loop with break statement inside the for loop. You can exit from within a FOR or WHILE loop using break. General break statement inside the for loop:

```
for I in 1 2 3 4 5
do
     statements1      #Executed for all values of ''I'',
     up
                      #to distater condition
     if (disaster-condition)
     then
         break               #Abandon the loop.
```

```
        fi
        statements3    #While good and, no
    disaster-condition.
    done
```

Early continuation with continue statement

To resume the next iteration of the enclosing FOR, WHILE or UNTIL loop, use continue statement.

```
    for I in 1 2 3 4 5
    do
        statements1  #Executed for all values of ''I'', up
                     #to a disaster-condition
        statements2
        if (condition)
        then
              continue   #Go to next iteration of I in the
                         #loop and skip statements3
        fi
        statements3
    done
```

8.3. while loop

Syntax:

```
        while [ condition ]
        do
                command1
                command2
                command3
        done
```

The loop is executed as long as given condition is true.

For example write script named **mul_Table** as follows:

```
if [ $# -eq 0 ]
then
      echo "Error - Number missing form command line argument"
      exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
      echo "$n * $i = `expr $i \* $n`"
      i=`expr $i + 1`
done
```

Run the above script as follows:

```
chmod +x mul_Table
sh mul_Table 7
```

Above loop can be explained as follows:

n=$1	Set the value of command line argument to variable n. (Here it's set to 7)
i=1	Set variable i to 1
	while [$i -le 10] This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
	echo "$n * $i = `expr $i \* $n`" Print multiplication table as 7 * 1 = 7 7 * 2 = 14 .... 7 * 10 = 70, Here each time value of variable n is multiply by i.
	i=`expr $i + 1` Increment i by 1 and store result to i. (i.e. i=i+1) **Caution:** If you ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than 10 and program will only output 7 * 1 = 7 ... E (infinite times)
	done Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated

Infinite loops

Infinite for while can be created with empty expressions, such as

```
while :
do
        echo "infinite loops [ hit CTRL+C to stop]"
```

```
done
```

Conditional while loop exit with break statement

You can do early exit with the break statement inside the whil loop. You can exit from within a WHILE using break.

In this example, the break statement will skip the while loop when user enters -1, otherwise it will keep adding two numbers:

```
while :
do
   read -p "Enter two numnbers ( - 1 to quit ) : " a
b
   if [ $a -eq -1 ]
   then
       break
   fi
   ans=$(( a + b ))
   echo $ans
done
```

Early continuation with the continue statement

To resume the next iteration of the enclosing WHILE loop use the continue statement as follows:

```
while [ condition ]
do
  statements1  #Executed as long as condition is true
          and/or,
               # up to a disaster condition if any
             statements2
  if (condition)
  then
        continue    #Go to next iteration of the loop and
                    #skip statements3
  fi
  done
```

9. Arrays

An array is a variable containing multiple values may be of same type . There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously. Array index starts with zero. To run all scripts related to arrays, use bash filename.

num[0]	num[1]	num[2]	num[3]	num[4]

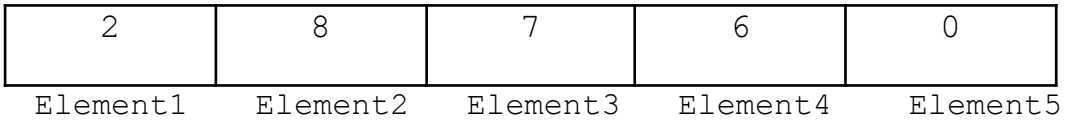

Figure: An Example of array

9.1 Declaring an Array and Assigning values

In bash, array is created automatically when a variable is used in the format like,

name[index]=value

- name is any name for an array
- Index could be any number or expression that must evaluate to a number greater than or equal to zero. You can declare an explicit array using declare -a arrayname.

```
Unix[0]='Debian'
Unix[1]='Red hat'
Unix[2]='Ubuntu'
Unix[3]='Suse'
or
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
echo ${Unix[1]}

output:
Red hat
```

To access an element from an array use curly brackets like ${name[index]}.

9.2 Print the Whole Bash Array

There are different ways to print the whole elements of the array. If the index number is @ or *, all members of an array are referenced. You can traverse through the array elements and print it, using looping statements in bash.

```
echo ${Unix[@]}
output:
Debian Red hat Ubuntu Suse
```

Referring to the content of a member variable of an array without providing an index number is the same as referring to the content of the first element, the one referenced with index number 0.

9.3 Length of the Bash Array

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

`${#arrayname[@]}` gives you the length of the array.

9.4 Length of the nth Element in an Array

${#arrayname[n]} should give the length of the nth element in an array

```
echo ${#Unix[3]} # length of the element located at index 3 i.e Suse
output:
4
```

9.5 Extraction by offset and length for an array

The following example shows the way to extract 2 elements starting from the position 3 from an array called Unix.

```
echo ${Unix[@]:3:2}
output:
Suse Fedora
```

The above example returns the elements in the 3rd index and fourth index. Index always starts with zero.

9.6 Extraction with offset and length, for a particular element of an array

To extract only first four elements from an array element. For example, Ubuntu which is located at the second index of an array, you can use offset and length for a particular element of an array.

```
echo ${Unix[2]:0:4}
output:
Ubun
```

The above example extracts the first four characters from the 2nd indexed element of an array.

9.7 Search and Replace in an array elements

The following example, searches for Ubuntu in an array elements, and replace the same with the word ‘SCO Unix’.

```
echo ${Unix[@]/Ubuntu/SCO Unix}
output:
Debian Red hat SCO Unix Suse Fedora UTS OpenLinux
```

In this example, it replaces the element in the 2nd index ‘Ubuntu’ with ‘SCO Unix’. But this example will not permanently replace the array content.

9.8 Add an element to an existing Bash Array

The following example shows the way to add an element to the existing array.

```
Unix=("${Unix[@]}" "AIX" "HP-UX")
```

In the array called Unix, the elements ‘AIX’ and ‘HP-UX’ are added in 7th and 8th index respectively.

9.9 Remove an Element from an Array

unset is used to remove an element from an array.unset will have the same effect as assigning null to an element.

```
unset Unix[3]
echo ${Unix[3]}
```

The above script will just print null which is the value available in the 3rd index. The following example shows one of the way to remove an element completely from an array.

9.10 Remove Bash Array Elements using Patterns

In the search condition you can give the patterns, and stores the remaining element to an another array as shown below.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora');
patter=( ${Unix[@]/Red*/} )
echo ${patter[@]}
output:
Debian Ubuntu Suse Fedora
```

The above example removes the elements which has the patter Red*.

9.11 Copying an Array

Expand the array elements and store that into a new array as shown below.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
Linux=("${Unix[@]}")
echo ${Linux[@]}
output:
Debian Red hat Ubuntu Fedora UTS OpenLinux
```

9.12 Concatenation of two Bash Arrays

Expand the elements of the two arrays and assign it to the new array.

```
Unix=('Debian' 'Red hat' 'Ubuntu' 'Suse' 'Fedora' 'UTS' 'OpenLinux');
Shell=('bash' 'csh' 'jsh' 'rsh' 'ksh' 'rc' 'tcsh');
UnixShell=("${Unix[@]}" "${Shell[@]}")
echo ${UnixShell[@]}
echo ${#UnixShell[@]}
output:
Debian Red hat Ubuntu Suse Fedora UTS OpenLinux bash csh jsh rsh ksh rc
        tcsh
```

It prints the array which has the elements of the both the array ‘Unix’ and ‘Shell’, and number of elements of the new array is 14.

9.13 Deleting an Entire Array

unset is used to delete an entire array.

```
unset Unix
echo ${#UnixShell[@]}
output:
0
```

After unset an array, its length would be zero as shown above.