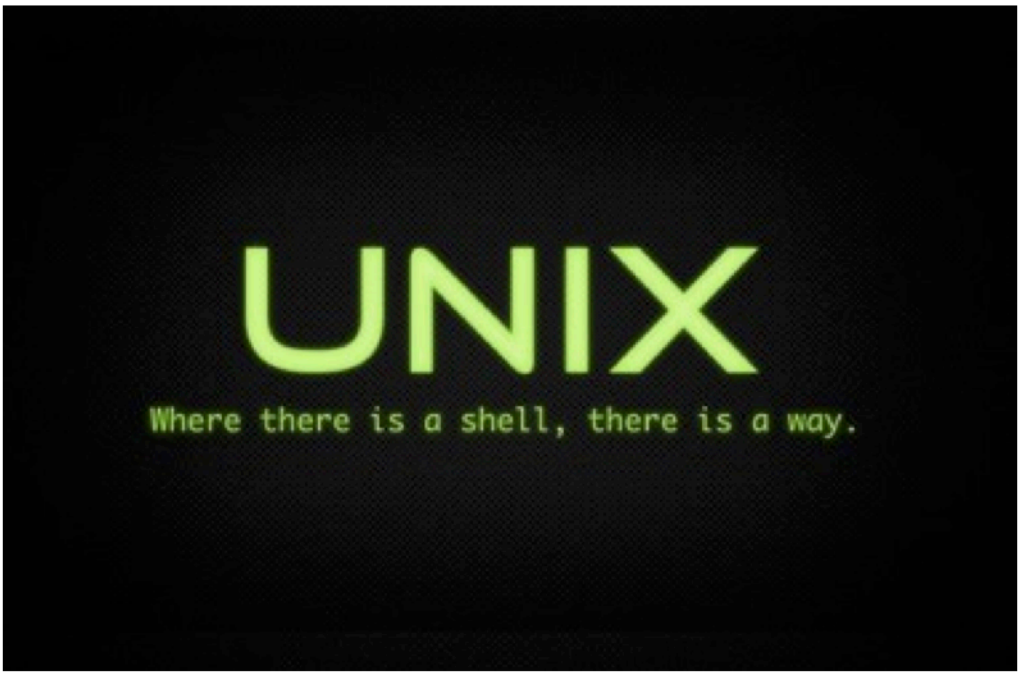
UNIX SHELL SCRIPTING



Compiled By

Mohammed Abdul Sami

sami.ma@gmail.com

ORACLE Certified Associate
Oracle Solaris 11
System Administrator

ITIL

redhat. CERTIFIED ENGINEER

bmcsoftware CERTIFIED ADMINISTRATOR
BMC CONTROL-M FOR DISTRIBUTED SYSTEMS

Unix Shell Scripting

Unix

Unix is a multitasking, multi-user computer operating system originally developed in 1969 by a group of employees at Bell Labs, including Ken Thompson, Dennis Ritchie.

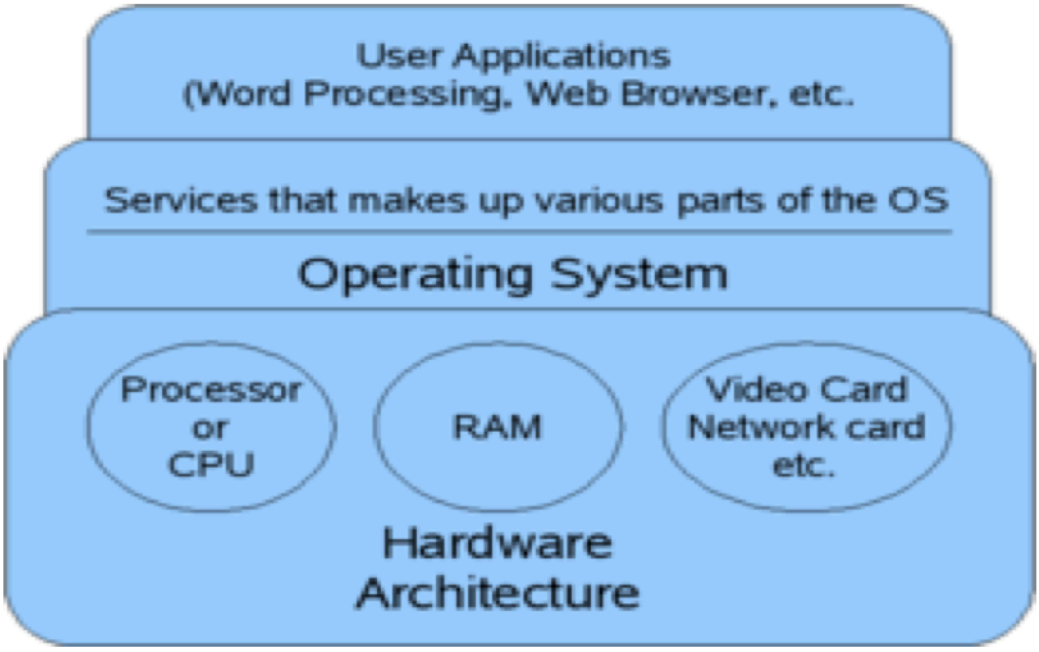
Unix trademark is now owned by The Open Group, an industry standards consortium.

Unix-like

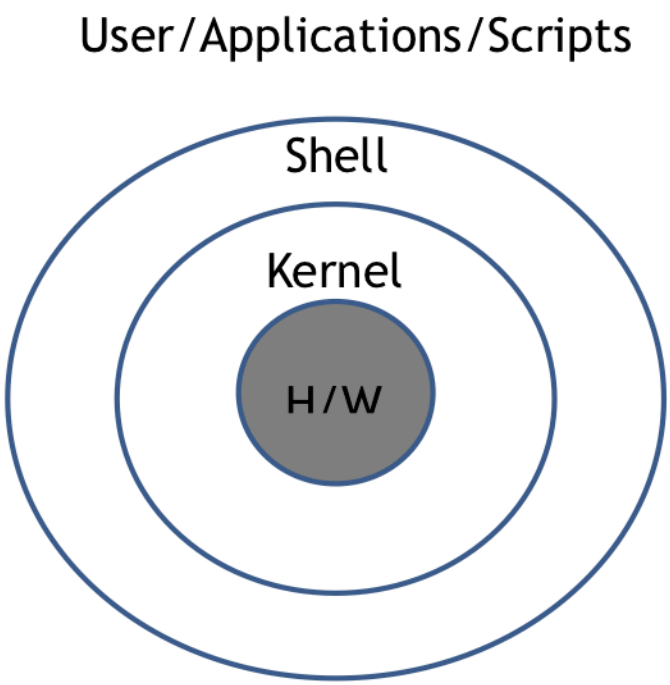
A Unix-like (sometimes referred to as UN*X or *nix) operating system is one that behaves in a manner similar to a Unix system, while not necessarily conforming to or being certified to any version of the Single UNIX Specification.

There is no standards for defining the term, and some difference of opinion is possible as to the degree to which a given operating system is “Unix-like".

OS Architecture



Unix Architecture



What is SHELL?

Shell is a UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters.

Functions & Role of SHELL

- Program Execution
- Environment Control
- I/O Redirection
- Pipe Line Hookup
- Wildcard Substitution
- Interpret Programing Languages

Type of Shells:

Unix / Linux OS offers a variety of shell environments to select from. Below are the popular Unix / Linux shells available.

- Borne shell (sh)
- Bourne again shell (bash)
- 'C' Shell (csh / tcsh)
- Korn shell (ksh)

Bourne Shell (sh)
The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977.

Bourne Again Shell (bash)
The "Bourne-again Shell," based on sh -- has become the new default standard. One attractive feature of bash is its ability to run sh shell scripts unchanged.

C – Shell (csh / tcsh)
Using C syntax as a model, Bill Joy at Berkeley University developed the "C-shell," csh, in 1978. Ken Greer, working at Carnegie-Mellon University, took csh concepts a step forward with a new shell, tcsh, which Linux systems now offer.

Korn Shell (ksh)
David Korn developed the Korn shell, or ksh, about the time tcsh was introduced. Ksh is compatible with sh and bash. Ksh improves on the Bourne shell by adding floating-point arithmetic, job control, command aliasing and command completion. AT&T held proprietary rights to ksh until 2000, when it became open source.

Vi Editor - brushup

Cursor movement

h - move left
j - move down
k - move up
l - move right
w - jump by start of words (punctuation considered words)
W - jump by words (spaces separate words)
e - jump to end of words (punctuation considered words)
E - jump to end of words (no punctuation)
b - jump backward by words (punctuation considered words)
B - jump backward by words (no punctuation)
0 - (zero) start of line
^ - first non-blank character of line
$ - end of line
G - Last Line (prefix with number - 5G goes to line 5)

Note:†Prefix a cursor movement command with a number to repeat it. For example, 4j moves down 4 lines.

Inserting / Appending

i - start insert mode at cursor
I - insert at the beginning of the line
a - append after the cursor
A - append at the end of the line
o - open (append) blank line below current line (no need to press return)

O - open blank line above current line
ea - append at end of word
Esc - exit insert mode
r - replace a single character (does not use insert mode)
J - join line below to the current one
cc - change (replace) an entire line
cw - change (replace) to the end of word
dd - delete (cut) a line
dw - delete (cut) the current word
x - delete (cut) current character

Cut and Paste
yy - yank (copy) a line
2yy - yank 2 lines
yw - yank word
y$ - yank to end of line
p - put (paste) the clipboard after cursor
P - put (paste) before cursor

Search/Replace

/pattern†- search for pattern
?pattern†- search backward for pattern
n - repeat search in same direction
N - repeat search in opposite direction
:%s/old/new/g - replace all†old†with†new†throughout file
:%s/old/new/gc - replace all†old†with†new†throughout file with confirmations

Exiting

:w - write (save) the file, but don't exit
:wq - write (save) and quit
:q - quit (fails if anything has changed)
:q! - quit and throw away changes

What is Shell Script

A Shell Script is computer program which consists of Unix / Linux shell commands in the order of their execution and which is designed to run under the Unix / Linux OS.

Writing Shell Scripts

A simple shell scripts is nothing but a list of commands in the order of their execution. For example the following shell script prints messages.

Note: As a best practice always use **.sh** as extension of shell script file name.

```
# My first  Shell Script
 clear
 echo  "From shell script"
 echo "Hello World"
 echo  "Bye  for Now"
```

To write and execute above code do the following.

Step 1: # **vi myfirst.sh**
Step 2: write the above code and save the script file and exit **vi**

Step 3: # **chmod +x myfirst.sh** (give execute permission)
Step 4: # **./myfirst.sh**

Note : To execute your script saved in current directory use **$./script_name** and if the script saved in a different directory the use full path **$/home/sami/myscripts/script_name.sh**

SHEBANG

Using SHEBANG we can tell OS to use a particular Shell to interpret the script. Shebang must be the first line of the shell script. Shebang consist of # (hash SHA) & ! (exclamation or bang).

#!/bin/bash -- tell OS that the following script should be executed in Bourne Again Shell

Consider the following script:

```
#!/bin/bash
# This script runs in bash shell
pwd
ls
```

LAB Exercise

Lets Write Some Shell Scripts

1) Writing a shell script to find biggest file in the directory.

```
#!/bin/bash
 ls –lSr|tail –n1
```

2) List out only user names with 'nologin' as default shell from /etc/passwd file

```
#!/bin/bash
 cat /etc/passwd|grep nologin|cut  -d':'  -f1
```

3) Write a script to do **messages & wtmp** log file cleanup from /var/log script works as root user

```
#!/bin/bash
# Log Cleanup  Tool
#  Works only as root user
Echo  "Initiating Log cleanup……"
cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Log files cleaned up."
```

Try These Scripts:

1) Write a shell script to find 5 big file in directory
2) Write a shell script to find system uptime and user names who are all users logged in
3) Write a shell script to find out number of user logged in (like 2 users etc)

Shell Variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. The shell enables you to create, assign, and delete variables.

Rules:

- Must begin with Letter or underscore ('_')
- Case sensitive
- No Special characters allowed in variable name except underscore ('_')

Type of Variables

SYSTEM Variables: Created by SHELL itself. These variables are useful to setting system environment. Eg : PATH, DISPLAY etc

User Defined Variables (UDV): Created and managed by user by user.

Local Variables – A local variable is a variable that is present within the current instance of the shell script. It is not available to programs that are started by the shell.

Environment Variables – An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

User Defined Variables:
Creating a UDV: A UDV variable can be created simply by assigning a value (or NULL) to the variable or by using "declare" command.

Eg:
```
NUM1=10
NAME="Sami"
declare  NUM1=10   NUM2
```

Assigning Value: Variable can be assigned a value using "=" without spaces either side.

Eg :
```
NUM1=10
NAME="Sami"
```

Accessing the value of Variable: Variable's value can be accessed by using '$' followed by variable name.

Eg:
```
NAME2=$NAME
echo $NUM1
```

Delete Variable: A Variable can be deleted using "**unset**" command

Eg:
```
unset  NAME
unset  NUM1
```

Making Variable Readonly: Readonly variable's value cannot be changed or that variable cannot be deleted

Eg:
```
readonly  NAME
```

The **declare** or **typeset** statements can be used to declare variables and set certain characteristics to them

Declare options:

The following options set the characteristics of variables.

-i Integer type variable

-r Readonly variable

-a Array type of variable

-l Upper case letters will be converted to lower case letter when strings stored in variable

-u Lower case letters will be converted to upper case letters when strings stored in variable

-x Declares a variable will be available for exporting as environment variable outside the script

Example Of Declaring an Integer Variable

```
declare -i   NUM1=10
```

Example of Declaring an Read-only Variable

```
typeset -r  NUM2=30
```

Example of Declaring an Array

```
declare -a  AR1=(10 20 30 40 50)
```

creates a single dimensional array with 5 elements and assign values 10 to 50 for each cell
Like **declare** statement **typeset** statement can be used to declare variables with specific characteristics

Special Variables

Variable	Description
$0	The filename of the current script.
$<n>	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.

$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

echo command

display text or value of variable.

USAGE: **`echo [options] [string, variables...]`**

Options
`-n` Do not output the trailing new line.
`-e` Enable interpretation of the following backslash escaped characters in the strings:

- **`\a`** alert (bell)
- **`\b`** backspace
- **`\c`** suppress trailing new line
- **`\n`** new line
- **`\r`** carriage return
- **`\t`** horizontal tab
- **`\\`** backslash

Example : **`echo -e "Hello World \n Welcome"`**

read command

USAGE : **`read -p "Prompt Message" <Variable Name>`**

Allows user to assign the value to a variable by inputting from keyboard

Examples:

A Script to ask name and print the given name

```
#!/bin/sh

read -p "Enter Your Name : "  PERSON
echo "Hello, $PERSON"
```

Quoting

Quoting is used to accomplish two goals:

1. To control (i.e., limit) substitutions and
2. To perform grouping of words.

Strong Quotes: Single Quotes (‘ ‘) . Enclosed text will be left alone with no variable or command substitution.
Eg: **`echo 'Is your home directory $HOME?'`**

Weak Quotes: Double Quotes [“ ”]. Enclosed text will be expanded if it contains variables.

Back Ticks [` `]: Enclosed text assumed as command and will be executed and the output returned.

Eg: **`HRS=`date +%h``**

eval Command

The **eval** command can be used execute any unix command which is saved in the variable

Example

```
$  X="date"
$ EVAL $X
```

Shell Arithmetics

These are the following arithmetic operators supported by bash Shell.

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
++	post increment eg: $((X++))
- -	post decrement eg $((X—))
**	exponent $((X ** 2))

In bash shell Integer Arithmetic is done using a external program **expr, let** statement or bracket expansion. Floating point arithmetic can be done using **bc** command.

Eg1:

```
#!/bin/bash
N1=10
N2=20
echo   `expr   $N1 + $N2`
```

Eg2:

```
#!/bin/bash
N1=10
N2=20
N3=`expr   $N1 + $N2`
echo "Sum is $N3"
```

Note: Spaces around arithmetic operator is mandatory.

The `let` statement
`let` statement can be used instead of **`expr`** for integer arithmetic.
for example the above code is rewritten using **`let`**

```
#!/bin/bash
N1=10
N2=20
let N3=$N1+$N2`
echo “Sum is $N3”
```

The third way of doing arithmetic is using bracket expansion

```
$((expression))
```

Example

```
#!/bin/bash
n=5
m=10
ans=$(( $n + $m))
echo "$n + $m = $ans"
```

Floating Point Arithmetic

Shell script are not meant for complex Arithmetic calculations but they support limited floating point operation using **`bc`** command. **`bc`** is an interactive command can be invoked from command prompt by typing **`bc -l`** then it allows user to type an expression using floating point numbers or integers and press **`ENTER`** key to perform calculation and print the result.

ex:

```
$ bc -l
2.1 + 2.2 <enter>
4.3
```

exit **`bc`** shell using **`CTRL+D`**

`bc` command can be used in shell scripts non interactively by piping a valid arithmetic expression to **`bc -l`** and then assigning the result to a variable.

Eg:

```
#!/bin/bash
N1=2.1
N2=2.2
N3=`echo  “$N1 + $N4”|bc -l`
echo “Sum is $N3”
```

Lab Exercise

1. Write a shell script to read 2 numbers and print them
2. Write a shell script to read 2 numbers and print their sum
3. Write a shell script to read 2 numbers find the difference
4. Write a shell script to read 2 numbers and find product
5. Write a shell script to read 2 numbers and find remainder
6. Write a shell script to create integer variable and try to read text to it.
7. Write a shell script to read name, marks of 3 subjects and print the total marks and percentage assuming max marks as 50.

Comparison Operators

Operator	Description
-eq	Checks if the value of two operands are equal or not, if yes then condition becomes true.
-ne	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
-gt	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
-lt	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
-le	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

String Comparison Operators

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
-z	Check if the string is not null
-n	Checks if the string is null

Boolean Operators

Operator	Description
!	This is logical negation. This inverts a true condition into false and vice versa.
-o	This is logical OR. If one of the operands is true then condition would be true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.

File Test Operators

Operator	Description
-d file	Check if file is a directory if yes then condition becomes true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.
-r file	Checks if file is readable if yes then condition becomes true.
-w file	Check if file is writable if yes then condition becomes true.
-x file	Check if file is execute if yes then condition becomes true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.
-e file	Check if file exists. Is true even if file is a directory but exists.
-O	Checks if the file owned by current user
-G	Checks if the file belongs to current user's group
-N	Checks if the given file modified since it was read let time
f1 -nt f2	checks if file f1 is newer than file f2
f1 -ot f2	checks if the file f1 older than file f2
f1 -ef f2	Check if both the files are hard links

String Manipulation

String Length

```
${#string}
```

Example

```
# X="date"
# Y=${#X}
# echo $Y
```

Extracting string from a position till end

```
${string:pos}
```

Example

```
var="Welcome"
echo ${var:3}
```

Extracting a Substring

```
${string:pos:No of Chars}
```

Example

```
var="Welcome"
echo ${var:1:3}
```

Find and replace string

${string/pattern/replacement}

Example

```
$ X="Shell Programming"
$ Y=${X//Programming/Scritpting}
$ echo $Y
```

Decision Making

Decision making helps in branching the code depending on certain condition such as if the condition is true a course of action is selected otherwise another course of action is selected.

Shell Script provides 2 ways to make decisions within the script.

The **`if...elif……else….fi`** statements
The **`case...esac`** statement

Example of decision Making Using **`if`** condition

```
#!/bin/bash
# This program is to  familiarize with IF statement
 echo –e "Script to  find biggest number among 2 given numbers\n"
 read –p "Enter first number : " N1
 read  -p "Enter second number : " N2
```

```
if  [ $N1 –gt  $N2 ]; then
    echo –e “\n\n  First number  $N1  is greater than second number $N2”
 else
 echo –e “\n\n  First number  $N1  is smaller than second number $N2”
 fi
```

Example 2: input a file or directory name and check if that file or directory exist

```
#!/bin/bash
# This script checks if the given file or directory exist or not
 read –p “Enter a File / Directory Name : “ FN
if  [ -e  $FN ]; then
    echo –e “The given File or Directory Exists”
 else
 echo –e “\a\a The given File or Directory Does Not Exists”
 fi
```

The case...esac Statement

case .. esac is simplified version of if condition can be used when a fixed set of values are expected for a particular variable and depending on the value a course of action needs to selected.

Example of **case .. esac**, assuming a script to calculate bill amount for a car rental company where depending on the model of car different rates are applied.

```
#!/bin/bash
read -p "Enter Car type BMW, LIMO, TAYOTA " CAR
case $CAR in
        BMW)
             rate=50
             ;;
        LIMO)
             rate=70
             ;;
        TAYOTA)
             rate=40
             ;;
        *)
             echo invalid choice
             ;;
esac
read -p "Enter Kilometers Traveled : " KM
AMT=`expr $rate \* $KM`
echo "Bill Amount is  $AMT"
```

Lab Exercise

1. Write shell script to read a integer number and find out if the number is odd or even
2. Write shell script to read 2 numbers and find biggest one
3. Write shell script to read 3 numbers and find biggest one among them
4. Write a shell script to read filename and find following info
    a) If the given file exist
    b) If exist, if it is a file or directory
    c) If file, whether its size is non zero
    d) If file, whether it has execute permission
    e) If directory then print message that "Given Name is Directory'

Loops

Loops helps us to repeat a block of code several times.
They are 4 types of loops which can be used in Shell scripting and they are

- The while loop
- The for loop
- The until loop
- The select loop

Nesting Loops

All the loops support nesting concept which means you can put one loop inside another a similar or a different loops. This nesting can go to any depth based on your requirement.

`while` loop

The **`while`** loop enables user to execute a set of code repeatedly as long as the condition remains true.

Syntax

```
While [   <Condition>   ]
do
Statement(s) to be executed if condition is true
Done

Eg:

#!/bin/bash
a=0
while [ $a -lt 10 ]
do
echo $a
let a=$a+1
done
```

`While` loop with **`read`** statement repeats the loop until **`EOF`** reading single line for each iteration.
Syntax

```
while  read  <var>
do
Statement(s) to be executed
Done <  <filename>
```

Example

```
#!/bin/bash
while read LINE
do
       echo  $LINE
done  < /etc/passwd
```

Using **:** to always return true condition with While loop

Syntax

```
while :
do
Statement(s) to be executed
Done
```

until loop

The **until** loop enables you to execute a set of code repeatedly as long as the condition remains false.

Syntax

```
Until  [   <condition>   ]
 do
Statement(s) to be executed  as long as condition is false
Done
```

Example
Here is a simple example that uses the **until** loop to display the numbers zero to nine –

```
#!/bin/bash
a=0
until [  $a -gt 10 ]
do
echo $a
a=`expr $a + 1`
done
```

for loop

for Loop iterates for each item given in the list

Syntax

```
for var in <list of values (items) separated  by space>
do
Statement(s) to be executed for every item in the list.
Done
```

Example 1

```
#!/bin/sh
for  N in 0 1 2 3 4 5 6 7 8 9
do
echo $N
Done
```

for loop can iterate for each items in given set. This set can be a set of numbers, words or list of filenames from a directory etc.

Below example uses **for** loop to iterate for each file present in the current directory.

Example 2

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

```
for FILENAME in `ls`
do
  echo $FILENAME
done
```

C-Style for loop can be used in Shell Scripting

Syntax

```
for  ((  initialization ;  <condition> ;  increment ))
{
   Statements to be executed
}
```

Example

```
for ((  i = 1 ; i  <= 10 ; i++ ))
{
echo $i
}
```

select loop

The select loop provides an easy way to create a numbered menu system from which users can select options from. It is useful when you need to ask the user to choose one or more items from a list of choices.

```
select var in OPT1 OPT2 ... OPTn
do
    Statement(s) to be executed for every word.
Done
```

Let us rewrite the script used in **case .. esac** where instead of inputting the car model we present user with menu to select a car model.

```
#!/bin/bash

select CAR in  BMW LIMO TAYOTA QUIT
do
      case $CAR in
                BMW)
                    rate=50
                    ;;
                LIMO)
                    rate=70
                    ;;
                TAYOTA)
                    rate=40
                    ;;
                QUIT)
                    echo  "Bye…"
                    exit
                    ;;
                *)
                    echo invalid choice
                    continue
```

```
                ;;
        esac
        read -p "Enter Kilometers " KM
        AMT=`expr $rate \* $KM`
        echo "Bill Amt is  $AMT"
done
```

Loop Control Statements

Loop control statements help in altering the iterations. There are 2 loop control statements

Break
Continue

The **break** statement
The **break** statement is used to terminate the execution of the entire loop

Syntax
Break

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.

Syntax
continue

Arrays

An array variable can hold multiple values at the same time. Arrays provide a method of grouping a set of data. Instead of creating a new variable for each value, a single array variable can be used to stores multiple values.

Initializing an Array

```
array_name=(value1    value2    ... valueN)
```

Accessing a single element
${array_name[index]}
Eg :

echo ${array_name[0]} it will print the first element from array
echo ${name[*]} prints all elements from the array
echo ${array_name[@]} prints all elements from the array same as above

Lab Exercise

1. Write a shell script to read 2 numbers and display a menu to let user select any of the 5 arithmetic operations (+,-,*,/,%) and do the calculation accordingly also display the operation performed and result.
2. Write a menu script to give user a choice perform copy, rename or delete operation on given filename. Read filename and destination (2 variables).
3. Write a script using while loop to print even numbers between 0 to 50
4. Write script using while loop to print the sum of numbers between 100 and 150
5. Write script using while loop to print the below pattern of numbers

```
1
12
123
1234
12345
```

6. Write a script using loop to print the below pattern

```
1
22
333
4444
55555
```

7. Write a script to print the below pattern using loops

```
*
**
***
****
*****
```

8. Write script using until loop to print ODD numbers between 0 to 50
9. Write script using until loop to print the sum of numbers between 100 and 150
10. Write a Script using any of the loops to print the factors of given number
11. Write a script using any loops to generate the following number pattern

```
1 1 2 3 5 8 13 21 34
```

12. Write a script using any loops to print the factorial of any given number.

Ex: if N is the number then factorial of N is 1 x 2 x 3 x 4 x …. x N

13. Write a while loop to read a text file and print number of words each line has.
14. Write a script to count the number of files having execute (x) permission in the given directory using for for loop
15. Write a shell script to delete all files from /tmp directory which we own
16. Write a shell script to determine how many plain text files exists in a given directory
17. Write a shell script to find files older than 7 days from a particular directory and the move & gzip those files to ~/backup directory.
18. Write a script to input a username and validate if the given username exists in system and if exist check if the user is currently logged in
19. Write a script to backup and zip /var/log/messages file if successful cleanup the messages log file (on debian/ubuntu systems /var/log/syslog)
20. Write a script to test if the given file is plain text file. if so, count all the blank lines.

21. Write a script to test if the given file is plain text file and contain more than 10 lines. if so, copy lines 5 to 9 to another file.
22. Write script to count how many empty files are the in home directory.

Parameters

Arguments are passed from the command line into a shell program are called parameters. Each parameter can used within the shell script by using special variables $1 to $9 each corresponding its position as supplied in the command line. After 9th parameter we can access 10th using ${10}, ${11} .. etc

Eg: Lets do a simple adding 2 number script where we pass those 2 numbers as parameters rather than doing a read or assigning value directly. Save the following code in **parm.sh** file.

```
#!/bin/bash
Z=`expr   $1  +  $2`
 echo  "Passed numbers are  $1   and   $2"
 echo "Sum is  $Z"
```

Execution

```
$ ./parm.sh     20  30
```

Note: Parameters after 9 can be accessed by using flower braces like **${n}**

Parameter Shifting

Within a script supplied value of positional parameters can be shifted towards left. For example if 2 parameters ($1 and $2) are passed then shifting results in moving the value of $2 to $1.

Eg:

```
#!/bin/bash
shift
Z=`expr   $1  +  $2`
 echo  "Passed numbers are  $1   and   $2"
 echo "Sum is  $Z"
```

Execution

```
$ ./parm.sh     20  30
```

Execution

```
$ ./parm.sh     20  30  40
```

Changing the Values of Parameter Variables

Eg:

```
#!/bin/bash
echo  "Passed Values are  $1   and   $2"
set Delhi  Pune
echo  "Changed values from Script are  $1  &   $2”
```

Execute the above script

```
$ ./parm.sh  100 200
```

Special Variables related to Parameters

$* - It stores the complete set of positional parameters as a single string.

$@ - Same as $*. Except that quoted parameters with space within treated as single unit.

$# - It is set to the number of arguments supplied. This lets you design scripts that check whether the right number of arguments have been entered.

$0 - Refers to the name of the script itself.

Advance Parameters

it will be possible to pass the parameters more professionally like built-in scripts or commands where the parameters can be passed as single letter preceded by a - and optionally a argument value (like we use **-l** with **ls** command to get long list eg: **ls -l**).
getopts can be used in the script to parse those parameters one by one each time getopts called after last parameter it returns a non zero return code.

getopts sets 2 local shell variables

OPTIND — it returns the next parameter number.
OPTARG — It is set to the passed parameter's argument value.

syntax

```
getopts   ":list of expected parameters optionally followed by a :
or ::"
```

where single **:** indicates that the preceding parameter has argument value and a **::** indicates argument value is optional.

example **gopts.sh**

```
#!/bin/bash
while  getopts ":ac:"  OP
do
    case $OP in
        a)
            echo "This is options  -a"
             ;;
       c)
            echo "This is options c and its values is $OPTARG"
             ;;
       *)
             echo "Invalid options"
             ;;
esac
```

Execution

```
$ ./gopts.sh  -a  -c 10
```

Functions in Shell Script

Big and complex scripts can be broken into smaller units which performs specific task. Following benefits of Function.

- Simplifies Code
- Allows reuse of code

- Easy to maintain

A Function can optionally take arguments and return values as well

Usage:

```
#!/bin/bash
myfunction()
{
Stments
Stment
}
#Main Script
Other statemnts
myfunction
Other stmnts
```

Execution of the script starts from Main Script and the code written within the functions is executed when the function is called.

Example of the usage of function.

```
#!/bin/bash
# A simple script with a function...

add_a_user()
{

 echo "Adding user $1 ...“
 useradd $1
 echo $2|passwd --stdin $1
 echo "Added user $1  with pass $2"
}

# Main body of script starts here
echo "Start of script..."
add_a_user bob  b0b123
add_a_user scott  Tiger
echo "End of script…"
```

In the above script function **add_a_user** is called twice with 2 arguments a username and a password from the main script each time it will add the given username given as first argument and set the password to the string given as second argument.

Scope of Variable

It is surprise to know that there is no scoping in shell script for UDVs. A variable assigned value is available in all the functions and main script as well. If the value is changed any of the functions or main scripts then same is reflected everywhere.

Consider Following Example:

```
#!/bin/bash

myfunc()
{

  x=2
}
```

```
### Main script starts here

x=1
echo "x before calling function is $x"
myfunc
echo "x after calling function is $x"
```

Local Variables

However a variables visibility can be confined to that function by declaring them as **local** variables.

Eg

```
#!/bin/sh

myfunc()
{

  local x=2
}
### Main script starts here

x=1
echo "x before calling function is $x"
myfunc
echo "x after calling function is $x"
```

NOTE: local variables can be declared within functions only

Return Value

Functions can return values that will be assigned to the special variable **$?** in the calling script.

Eg

```
#!/bin/bash
myfunc()
{
 local SUM
 SUM=`expr  $1 + $2`
 return $SUM
}

#Main Function
read -p "Enter a Number " X
read -p "Enter second number " Y
myfunc $X  $Y
echo "Sum is  $?"
```

Library of Functions

It is often easier to write a "library" of useful functions, and source that file at the start of the other scripts which use the functions.

Lets say we have written all our useful functions in file called as **myfunctions.sh** and stored this file in **/etc/** directory.

Write the following line at beginning of any script to call any of these functions.

```
source /etc/myfunctions.sh
```

Debugging the Script

Debugging helps us in finding where the error is occurring and what were the values of variables at that time. Debugging can be enabled by putting –x after shebang for example

```
#!/bin/bash -x
```

Or using **set** command immediately after shebang

```
#!/bin/bash
set –x
```

Note: Remove the **-x** from the script once debugging is completed

Exiting on Error

Shell scripts don't exit on error but continue to execute next line. This behavior can be altered using **-e** with shebang or using set command. For example

```
#!/bin/bash –e
```

Or

```
#!/bin/bash
 set -e
```

Signals

Signals are software interrupts sent to a running process to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

Default Actions

Every signal has a default action associated with it. The default action of a signal is the action that a script or program performs when it receives that signal.

Some of the possible default actions are –

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called core containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

Some of the commonly used signals.

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).

Note: Use **`kill –l`** command to see all available signals.

Sending Signals

There are several methods of delivering signals to a program or script. One of the most common is for a user to type **`CONTROL-C`** or the **`INTERRUPT`** key while a script is executing.
When you press the **`Ctrl+C`** key a signal named **`SIGINT`** is sent to the script and as per defined default action script terminates.

The other common method for delivering signals is to use the **`kill`** command whose syntax is as follows –

```
$ kill   –<signal>    <pid>
```

Signal Trapping

Traps allows us to change the default action of signals.

Usage

```
trap   <command for alternate action>   <Signal(s)>
```

Eg: Below Script changes the default action of signal **`SIGINT (CTRL+C)`** and set it to a **`echo`** command and **`exit`** command.

```
#!/bin/bash
trap 'echo someone pressed crtl-c; exit' SIGINT
x=0
while [ $x -lt 10 ]
do
x=0
done
```

Ignoring Signals

If the command listed for **`trap`** is null, the specified signal will be ignored when received.

For example, the command –

```
$ trap '' 2
```

You can specify multiple signals to be ignored as follows

```
$ trap '' 1 2
```

Resetting Traps

Default action of the signal can be restored using **trap** command with out specifying command

```
$ trap 1 2
```

The Initialization Scripts

The effect of assigning values to variables, defining aliases and using set options is applicable only for the login session; they revert to their default values when the user logs out. To make them permanent, use certain startup scripts. The startup scripts are executed when the user logs in. Below are the list of startup scripts SHELL executes after completing the login

- **.profile** (Bourne shell)
- **.profile** and **.kshrc** (Korn shell)
- **.bash_profile** (or **.bash_login**) and **.bashrc** (Bash)
- **.login** and **.cshrc** (C shell)

Immediately after user login **/etc/profile** is executed and after that it looks for the following files

- **~/.bash_profile**
- **~/.bash_login**
- **~/.profile**

They are searched in that order if none exist then **~/.bashrc** is executed. **~/.profile** file is executed only once at the time of login.

The rc Files

The rc files are designed to be executed every time a separate shell is created.

no rc file in Bourne,
bash uses **~/.bashrc**
korn uses **~/.kshrc**

Environment Variables

Bash uses **BASH_ENV=$HOME/.bashrc**
Korn uses **ENV=$HOME/.kshrc**

Note: Bash executes rc file only when a subshell is created an entry can be made in profile to execute it at the time of login.

.~/.bashrc

Korn executes RC file at the time of login if ENV variable is set.

Logout Script

~/.bash_logout

If it exist bash executes this script when user logs out

Good Practices of Shell Scripting

- Use good indentation
- Provide usage statements
- Use sensible commenting
- Provide comments that explain your code
- Exit with a return code when something goes wrong
- Use functions rather than repeating groups of commands
- Give meaningful names to variables
- Check that sufficient arguments are passed
- quote all parameter or variable expansions
- Check if needed files actually exist
- Always use **`set –e`** at the beginning of the script