Shells and Shell Programming - BASH
Chapter - 5

Presentation from Uplatz

Contact Us: https://training.uplatz.com/
Email: info@uplatz.com
Phone: +44 – 7836 212635

5. Shells and Shell Programming (BASH)

- Command line interpreters and SSH
- Variables in shell (Local and Global (export))
- Environment variables
- How to write the script?
- Quotes (Single and Double along with variables)
- Test commands or [expr]
- Operators (Arithmetic Operators, Increment and Decrement Operators, Relational Operators, Logical or Boolean Operators, String Operators, File Test Operators)
- Conditional execution (&& and ||)
- Conditional statements (if … fi, if … else … fi, if … elif … else … fi, nested if, case)
- Repetitive statements (for, while, until loops), loop control statements (break, continue) and Nested Loops
- Arrays
- Functions
- Local and Global Variables
- Executing other programs
- Command line arguments
- Command line options (getopts)
- Signal Handling (Default action, Handling Signals and Ignoring Signals)
- Commands such as kill, trap, shift
- Debugging (set)
- Utilities: bc, cmp, diff, uniq, paste, join, cut, tr, sed

Command Line Interpreter

- Its job is to get and execute the next command statement
- Request are given to the OS to deal with process creation, I/O handling, memory management and Networking
- Shell is a command line interpreter

What is a shell?

- Shell is a program.
- It acts as a command line interpreter.
- Shell verifies the command usage.
- Shell provides the basic programming capabilities.

Command Line Interpreters

- Shell: tools to execute user commands
- Called “shells” because they hide the details on the underlying operating system under the shell’s surface.
- Commands are input in a text terminal, either a window in a graphical environment or a text-only console.
- Results are also displayed on the terminal. No graphics are needed at all.
- Shells can be scripted: provide all the resources to write complex programs (variable, conditionals, iterations …)

What are Shell Scripts?

- In the simplest terms, a shell script is a file containing a series of commands.
- The shell reads this file and carries out the commands as though they have been entered directly on the command line.
- The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter.
- Most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

What is a shell script?

- Shell script is a text file holds commands line by line.
- All the commands can be submitted at one go for execution.
- Commands in the shell script executes one by one.
- Its convention to have extension for the shell script as .sh or .bash
- Comment in the script file beings with #

Well known Shells

- Bourne Shell (sh) – Obsolete
  - Traditional, basic shell found on Unix Systems, by Steve Bourne
- C Shell (csh) – Obsolete
  - Once popular shell with a C-line syntax
- TC Shell (tcsh) – Tab C Shell - Still popular
  - A C shell compatible implementation with evolved features (command completion, history editing and more …)
- Korn Shell (ksh)
- Bourne Again SHell (bash) – Most Popular
  - An improved implementation of sh with lots of added features too.

Resource file in BASH

- Resource Control or Run Commands or Run Control or Runtime Configuration
- The ‘rc’ suffix goes back to Unix's grandparent, CTSS (Compatible Time-Sharing System). It had a command-script feature called "runcom". Early Unixes used ‘rc’ for the name of the operating system's boot script, as a tribute to CTSS runcom.
- ~/.bashrc
  - Shell script read each time a bash shell is started
- You can use this file to define
  - Your default environment variable (PATH, EDITOR etc…)
  - Your aliases
  - Your prompt
  - A greeting message

Introduction to SSH

- SSH stands for Secure Shell
- SSH is a secure communication protocol that allows remote login, file transfer and port tunneling, normalized by RFC 4251, 4252, 4253 and 4254.
- OpenSSH suite includes ssh program that replaces telnet and rlogin, and scp which replaces rcp and ftp.
- OpenSSH has also added sftp and sftp-server which implement an easier solution for file-transfer.
- On Linux, the main implementation is OpenSSH, free suit of tools, with both the server and client programs.
- On Windows, Putty is one of the free SSH client available.

Installation and basic usage

- OpenSSH is available as a package in all GNU/Linux distributions.
- On Ubuntu, two packages are available
  - openssh-client, the client programs
  - openssh-server, the server program
- Connecting to an SSH server is as simple as

  $ ssh username@hostname #or

  $ ssh user@ip-address
- ssh will prompt for the user password and log in to the remote system.
- Files can be transferred using the scp client program
  - scp myfile1 myfile2 username@hostname:~/dest/directory/
  - scp –r mydirectory user@host:~/dest/
- ssh not only allows to connect to a remote host, but also allows remote execution of commands
  - ssh user@host ls
  - This is very useful in shell scripts

Configuration file

- SSH stores a configuration file in ~/.ssh/config
- It can be used to set global options, but also per-host options, like
  - Host openmoko
  - Hostname 192.168.0.202
  - User root
- Using these options, running “ssh openmoko” will connect automatically to IP 192.168.0.202 with the root login.

Variable types

- Shell (or Local) variables
  - Available only to the current instance of the shell.
  - Local variables after exporting would become environment variables but only visible to that shell.

- Environment
  - Available to the current instance and as well as to all child processes spawned by any type of shell.
  - Ex: Created variable in BASH and visible in any shell (bash or sh or csh or ksh or tcsh) that spawned.

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.
- Convention to have a variable in upper case.

```
_PERSON
VALUE
VAR_1
VAR_2
```

Shell or User defined variables

- Shells let the user define variables.
  - They can be reused in shell commands.
  - Convention: lower case names
  - Shell variables (bash)

```
$ projdir=/home/marshall/coolstuff
$ ls –la $projdir; cd $projdir
```

Environment variables

- Environment variables are the variables used by shell to determine certain values.
- Lists out all the environment variables with corresponding values.
- How to see them?

  $ printenv

  $ env|more

  - Lists all defined environment variables and their value.

  $ echo $<variable> as in echo $DISPLAY

  $ env|grep <variable>

Environment variables

- How to set or define them?
  - Variables that are also visible within scripts or executables called from the shell.
  - Convention: upper case names.
  - A variable can be set by a very simple syntax “VAR=value”
  - Once you set, you can export the variable using the syntax “export VAR”
  - Then only it becomes environment variables
  - Environment variables are inherited by all the processes spawned by the current process
  - Environment variables (bash)

    $ cd $HOME

    $ export DEBUG=1

    $ ./find_extraterrestrial_life (displays debug information if DEBUG is set)

Local variable to Environment variable

- Exports a variable to child shells
- Syntax: $ export variable

- With out Export

```
$ v1=10
$ echo $v1
10
$ bash
$ echo $v1

$ exit
exit
$
```

- With Export

```
$ v2=25
$ echo $v2
25
$ export v2
$ bash
$ echo $v2
25
$ exit
exit
$
```

- With out Export

$ OS=LINUX

$ cat export_test.sh
#!/bin/bash
echo "OS=$OS"
OS=UNIX
echo "OS=$OS“

$ ./export_test.sh

- With Export

$ export OS=LINUX

$ ./export_test.sh

Main standard environment variables

Used by lots of applications!

- LD_LIBRARY_PATH
  - Shared library search path
- DISPLAY
  - Screen id to display X (graphical) applications on.
- EDITOR
  - Default editor (vi, emacs…)
- HOME
  - Current user home directory
- HOSTNAME
  - Name of the local machine

- MANPATH
  - Manual page search path
- PATH
  - Command search path
- PRINTER
  - Default printer name
- SHELL
  - Current shell name
- TERM
  - Current terminal type
- USER
  - Current user name

PATH environment variables

- PATH
  - Specifies the shell search order for commands

  /home/guest/bin:/usr/local/bin:/usr/bin:/bin/:/usr/x11R6/bin:/bin:/usr/bin
- LD_LIBRARY_PATH
  - Specifies the shared library (binary code libraries shared by applications, like the C library) search oder for ld

  /usr/local/lib:/usr/lib:/lib:/usr/X11R6/lib
- MANPATH
  - Specifies the search order for manual pages

  /usr/local/man:/usr/share/man

PATH usage warning

It is strongly recommended not to have the "." directory in your PATH environment variable, in particular not at the beginning:

- A cracker could place a malicious ls file in your directories. It would get executed when you run ls in this directory and could do naughty things to your data.
- If you have an executable file called test in a directory, this will override the default test program and some scripts will stop working properly.
- Each time you cd to new directory, the shell will waste time updating its list of available commands.

Call your local commands as follows: ./test

PATH

- Environment variable PATH tells the shell to looks into corresponding path/s for running it.

  $ echo $PATH

- Appending a new directory to PATH.

  $ PATH=$PATH:~/bin

PWD and OLDPWD

- Environment variable PWD and OLDPWD represents current PWD and previous PWD.

  $ echo $PWD

  $ echo $OLDPWD

- Also executed as

  $ echo ~+

  $ echo ~-

How to write and execute script?

- Shebang construct tells the shell to execute the script using specific shell.

```
#!/bin/bash #or
#!/usr/bin/env bash
```

- It must appear as the first line of the shell script.

```
# my first script file
# first.sh
#!/bin/sh
pwd
date
```

- Executing the script

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

Or

```
$ bash first.sh
```

Writing Simple Shell Scripts

- Create a file with .sh extension

  $ touch myscript.sh

- Give it execute permissions

  $ chmod u+x myscript.sh

- Edit the file to write the required code

  $ vi myscript.sh

- After saving the file, execute it either saying

  $ ./myscript.sh or

  $ bash myscript.sh

Syntax of a Shell Script

- Always starts with the directive #!/bin/bash to indicate that this is a shell script

  #!/bin/bash
- Comment lines start with # character
  - # This is a comment line
- Any command will look like as if you executed it in the command shell
  - echo "Hello World!"
- It is a good idea to end the program with an exit value (0 for success, non-zero for error value)
  - exit 0
- Example Shell Script (open vi myscript.sh and type it)

  #!/bin/bash

  # This is a comment line

  echo "Hello World"

  exit 0
- How to see the exit value of a script

  $./myscript.sh

  $ echo $?

  0

  $

Quoting mechanism

- Metacharacters which have special meaning

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions $ ` \$ \` \" \\
Backslash	Any character immediately following the backslash loses its special meaning.
Backquote	Anything in between back quotes would be treated as a command and would be executed.

Quoting mechanism

```
echo 'To get variable data prefix it with $ symbol such as $variable'


dt=`date`
echo "Date is > $dt"
#echo Date is > $dt # This would perform redirection


dt=$(date)              # Command substitution (equivalent to back quote)
echo $dt


# To execute multiple commands in single line
$ ls;pwd;date
```

Meta Characters

```
#!/bin/bash


echo 'Welcome! Shell Programming‘
echo -e "Hi!\tHello!\nWelcome to Shell Scripting"
v1=5
echo 'v1 is $v1'
echo "v1 is $v1"


v1=`date`
echo $v1
echo "It's Friday"
echo "It\"s Friday"
```

```
#!/bin/bash

$ date;pwd;ps -f #Executes the commands sequentially

$ (date;pwd;ps –f) # Executes the commands in child/sub shell
```

Command - echo

- Prints a line on the console output.

```
$ echo 'Happy Shell Scripting with BASH!'
Happy Shell Scripting with BASH!
$ echo "Hello World!"
Hello World!


$ val=10
$ echo $val
$ echo 'Value is $val'
$ echo "Value is $val"
```

Command - read

- Reads a line from the console input into a variable

```
$ read mystr
Hello
$ echo $mystr
Hello
$
```

Sample variable I/O script

- Inputs name from console input and echo's on the console output.

```
#!/bin/bash
#var_io.sh
echo "What is your name?"
read name
echo "Hello, $name"
```

Read-Only variable

- Command ‘readonly’ can be used to make a variable read only.

```
water= "hot";
readonly water;
water= "cool"
#above line gives an error ☹
```

unset

- Command ‘unset’ can be used to remove a variable.

```
$ mylanguage= "bash"
$ echo $mylanguage
$ unset mylanguage
# verify below!!!
$ echo $mylanguage
```

PS1

- Primary String or Prompt String 1, is the prompt displayed by shell, a set of characters.

```
$ PS1= "Hi :> "
Hi :>
```

PS1

- Escape sequence characters that can be used in string of PS1

Escape Sequence	Description
\t	Current time, expressed as HH:MM:SS
\d	Current date, expressed as Weekday Month Date
\n	Newline.
\s	Current shell environment.
\W	Working directory.
\w	Full path of the working directory.
\u	Current user.s username.
\h	Hostname of the current machine.
\#	Command number of the current command. Increases with each new command entered.
\$	If the effective UID is 0 (that is, if you are logged in as root), end the prompt with the # character; otherwise, use the $.

- Make the changes permanent by adding it to .bashrc file

Operator - test

- Checks the relation between two expressions

```
$ test 10 -eq 10
$ echo $?
```

```
$ [ 10 -eq 10 ]
$ echo $?
```

```
$ ((10==10))
$ echo $?
```

Command - test

```
v1=1

v2=1


test $v1 -eq $v2

echo 1. $?

test $v1 == $v2

echo 2. $?


[ $v1 -eq $v2 ]

echo 3. $?

[ $v1 ==  $v2 ]

echo 4. $?


((v1==v2))

echo 5. $?
```

Shell operators

- Arithmetic Operators.
- Increment and Decrement Operators.
- Relational Operators.
- Logical or Boolean Operators.
- String Operators.
- File Test Operators.

Expression

```
$ expr 1 + 3
4
$
```

```
#!/bin/bash
res=`expr 1 + 2`
echo $res

v1=1
v2=2
res=`expr $v1 + $v2`
echo $res

v1=2
v2=3
res=$((v1+v2))
echo $res
```

```
#!/bin/bash
let v1=2 v2=3 res=v1+v2
echo $res

let res= "(3+4)*5"
echo $res

let res='(3+4)*5'
echo $res

res=$((2+3))
echo $res

res=$(( (3+6) * 5 ))
echo $res
```

Arithmetic Operators

expr exp-1 operator exp-2

- Evaluating an expression with arithmetic operators.
  - Say v1=10 & v2=20

```
#!/bin/bash
v1=10
v2=3

res=`expr $v1 + $v2`
echo "Result of +: $res"

res=`expr $v1 - $v2`
echo "Result of -: $res"

res=`expr $v1 \* $v2`
echo "Result of *: $res"

res=`expr $v1 / $v2`
echo "Result of /: $res"
```

Operator	Example
+	`expr $v1 + $v2` will give 30
-	`expr $v1 - $v2` will give -10
*	`expr $v1 \* $v2` will give 200
/	`expr $v2/ $v1` will give 2
%	`expr $v1 % $v2` will give 10

```
#!/bin/sh
v1=11
v2=2

sum=`expr $v1 + $v2`
echo "sum is $sum"
mul=`expr $v1 \* $v2`
echo "mul is $mul"
mul=$((v1*v2))
echo "mul is $mul"
```

```
$ result=`expr 15 + 20`
$ echo $result
```

```
$ result=$((12+24))
$ echo $result
```

```
$ let v1=12 v2=24 res=v1+v2
$ echo $res
```

Example

```
echo "Enter Principal amount "
read prin
echo "Enter number of years "
read nyears
echo "Enter rate of interest "
read rinterest

si=`expr $prin \* $nyears \* $rinterest / 100`

echo "Simple interest=$si"
```

Increment/Decrement Operators

```
#!/bin/bash

v1=10
((v1++))
echo $v1
((++v1))
echo $v1

v1=10
v2=$((v1++))
echo -e "$v1\t$v2"

v1=10
v2=$((++v1))
echo -e "$v1\t$v2"
```

```
#!/bin/bash

v1=10
let "v1 = v1 + 1"
echo
v2=20
let "v2 += 1"
v3=v4=30
let "v3++"
let "++v4"
echo -e "$v1\t$v2\t$v3\t$v4"
```

Pre/Post Increment Operators

```
#!/bin/bash

# pre_post_incr.sh
v1=10
echo $((v1++))
echo $((v1++))
echo $((++v1))

v1=100
let "v1++"
echo -n "$v1 "
echo
echo

v1=200
((v1++))
echo $v1
echo
```

```
v1=300
: $((v1++))
echo $v1
echo

v1=400
: $[ v1++ ]
echo $v1
echo
```

Try with Decrement Operators

Relational Operator

- Say, v1=10 & v2=20

Operator	Example
-eq	[$v1 -eq $v2] is false
-ne	[$v1 -ne $v2] is true.
-gt	[$v1 -gt $v2] is false
-lt	[$v1 -lt $v2] is true.
-ge	[$v1 -ge $v2] is false
-le	[$v1 -le $v2] is true.

```
#!/bin/bash
v1=10
v2=20

[ $v1 -lt $v2 ]
echo "1. $?"

[ $v1 -le $v2 ]
echo "2. $?"

[ $v1 -gt $v2 ]
echo "3. $?"

[ $v1 -ge $v2 ]
echo "4. $?"

[ $v1 -ne $v2 ]
echo 5. $?

[ $v1 -eq $v2 ]
echo 6. $?
```

Logical or Boolean operators

- Say, v1=10 & v2=20

Operator	Example
!	[! $v1 -lt 20 -o $v2 -gt 100] is false.
-o	[$v1 -lt 20 -o $v2 -gt 100] is true.
-a	[$v1 -lt 20 -a $v2 -gt 100] is false.

```
v1=10
v2=20

[ $v1 -le 10 -a $v2 -le 10 ]
echo $?

[ $v1 -le 10 -o $v2 -le 10 ]
echo $?

[ ! $v1 -le 10 -o $v2 -le 10 ]
echo $?
```

```
#!/bin/bash
v1=10
v2=20
v3=30

[ $v1 -le $v2 -a $v2 -le $v3 ]
echo 1. $?

[ $v1 -le $v2 -a $v2 -gt $v3 ]
echo 2. $?

[ $v1 -le $v2 -o $v2 -le $v3 ]
echo 3. $?

[ $v1 -le $v2 -o $v2 -gt $v3 ]
echo 4. $?

[ ! $v1 -gt $v2 ]
echo 5. $?

[ ! $v1 -lt $v2 ]
echo 6. $?
```

Operators Precedence and Associativity

```
$ res=`expr 10 + 20 + 30`
$ echo $res
60
$ res=`expr 4 + 5 \* 6`
$ echo $res
34
$ res=`expr 4 \* 5 + 6`
$ echo $res
26
$ res=`expr 100 / 2 \* 5`
$ echo $res
250
$
```

String Operators

```
s1="hi"
s2="UNIX"
s3=""
[ $s1 ]
echo $?
[ $s3 ]
echo $?
[ -z $s1 ]
echo $?
[ -z $s3 ]
echo $?
s4="hi"
[ $s1 = $s4 ]
echo $?
[ $s1 != $s4 ]
echo $?
```

String operators

1. Compare two strings (=, !=)
2. Whether string is empty or not (-z, str)

Say, str1= "hi" and str2= "unix"

Operator	Example
=	[$str1 = $str2] is false
!=	[$str1 != $str2] is true
-z	[-z $str1] is false
str	[$str1] is true

Uplatz

```
#!/bin/bash
s1="hello"
s2=""
s3=$s1

[[ $s1 = $s2 ]]
echo 1. $?

[ $s1 == "$s2" ]
echo 2. $?

[ "$s1"!= "$s2"]
echo 3. $?

[ "$s1" = "$s3" ]
echo 4. $?

[ $s1 != $s3 ]
echo 5. $?

echo 6. :$s1: :$s2: :$s3:
```

```
#!/bin/bash
# string_zero_ops.sh

s1="hello"
s2=""

[ -z $s1 ]
echo 1. $? # 1

[ -z "$s2"]
echo 2. $? # 0

[ $s1 ]
echo 3. $? # 0

[ $s2 ]
echo 4. $? # 1

[ "$s2"]
echo 5. $? # 1

echo :$s1: :$s2
```

```
#!/bin/bash
mystr="Shell Scripting"
echo ${#mystr}  # To find string length

substr=${mystr:6} # Extracting substring
echo $substr

substr=${mystr:6:5} # Extracting substring with required length
echo $substr

# Find and replace string values
replacestr=${mystr/Script*/(BASH) Scripting}
echo $replacestr

# Replaces only first occurence
newstr="Path of bash is /bin/bash"
replacestr=${newstr/bash/BASH}
echo $replacestr

# Replaces all occurences
newstr="Path of bash is /bin/bash"
replacestr=${newstr//bash/sh}
echo $replacestr
```

File check or test operator

- Several checks can be done on files/directories to test the existence, permissions etc.
  - -f checks if the file exists or not and if it is a regular file
  - -d checks if the file exists and is a directory
  - -r checks if the file exists and is readable
  - -w checks if the file exists and is writable
  - -x checks if the file exists and is executable
    - if [$DATA && -f $OUTFILE]; then
  - For a complete list of possible checks, see man sh or man bash

File check or test operators

- Let file="poem.txt" with the following permissions

```
-rw-rw-r-- 1 trainer trainer 138 Mar  3 17:49 poem.txt
```

opr	Description	Example
-b	file Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c	file Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d	file Check if file is a directory if yes then condition becomes true.	[-d $file] is false
-f	file Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-p	file Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t	file Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-r	file Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w	file Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x	File Check if file is executable if yes then condition becomes true.	[-x $file] if false.

File Test Operators

```
myfile="poem.txt"

# Ensure you have the file, poem.txt, with certain text


[ -f $myfile ]

echo $?

[ -d $myfile ]

echo $?

[ -w $myfile ]

echo $?

[ -x $myfile ]

echo $?

[ -s $myfile ]

echo $?
```

```
#!/bin/bash

#filetest_ops.sh
mydir=`pwd`
touch emptyfile.txt
printf "This is file1.txt\nLine 2 of file\nEnd of file\n" > file1.txt
myfile="./emptyfile.txt"
f1="./file1.txt"

[ -d $mydir ]
echo 1. $? # 0
[ -f $mydir ]
echo 2. $? # 1
[ ! -f $mydir ]
echo 3. $? # 0
[ -f $myfile ]
echo 4. $? # 0

[ -d $myfile ]
echo 5. $? # 1
[ -s $myfile ]
echo 6. $? # 1
[ -s $f1 ]
echo 7. $? # 0
[ -d $f1 ]
echo 8. $? # 1
```

```
#!/bin/bash

#filetest_blockchar_ops.sh
# Ensure that the below block device file should exist
bdevfile="/dev/sda6"
cdevfile=`tty`

[ -b $bdevfile ]
echo 1. $? # 0
[ -c $bdevfile ]
echo 2. $? # 1
[ -d $bdevfile ]
echo 3. $? # 1
[ -f $bdevfile ]
echo 4. $? # 1

[ -b $cdevfile ]
echo 5. $? # 1
[ -c $cdevfile ]
echo 6. $? # 0
[ -d $cdevfile ]
echo 7. $? # 1
[ -f $cdevfile ]
echo 8. $? # 1
```

Conditional Execution (&&, ||)

```
$ false &&echo "Hi"
$ true &&echo "Hi"
Hi
$ true || echo "Hi"
$ false || echo "Hi"
Hi
$ v1=10
$ v2=20
$ [ $v1 -le 20 ] &&echo "hi"
hi
$ echo $?
0
$ [ $v1 -gt 20 ] &&echo "hi"
$ echo $?
1
$
```

Note: && and || evaluate the second argument only when needed

Decision making

- Linux or Unix Shell supports conditional statements to perform actions based conditions

```
if...else statements

case...esac statement
```

Condition Checks - if

- The if construct checks if a particular check is successful or not
  - Example: Check if the input arguments are provided or not

```
if [ $# -gt 0 ]; then
        echo "Input arguments are given"
else
        echo " No input arguments"
fi
```

- The elif keyword is used for a sequence of condition checks.
- Logical operations can be done on multiple conditions to derive complex condition checks.
  - !, &&, ||

if...fi statement

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
fi
```

if... else...fi statement

```
if [ expression ]
then
  Statement(s) to be executed if expression is true
else
  Statement(s) to be executed if expression is not true
fi
```

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

```
if [ expression 1 ]
then
  Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
  Statement(s) to be executed if expression 2 is true
else
  Statement(s) to be executed if no expression is true
fi
```

- case … esac statement

```
case expr in

pattern1)

Statement(s) to be executed if pattern1 matches

;;

pattern2)

Statement(s) to be executed if pattern2 matches

;;

pattern3)

Statement(s) to be executed if pattern3 matches

;;

esac
```

```
#!/bin/bash
#conditional_statements_if.sh

echo -n "Enter any number: "
read num

echo "number is $num"

if [ $num -ge 0 ]
then
  echo "You entered +ve number or zero"
fi
```

Conditional Statement – if Sample

```
echo "Enter hours "
read hrs
echo "Enter minutes"
read mins
mins=`expr $mins + 1`

if [ $mins -gt 59 ]
then
        mins=0
        hrs=`expr $hrs + 1`
fi

if [ $hrs -gt 23 ]
then
        hrs=0
fi
echo "$hrs:$mins"
```

```
#!/bin/bash
#conditional_statements_if_else.sh

echo -n "Enter any number: "
read num

echo "number is $num"

if [ $num -ge 0 ]
then
   echo "You entered +ve number or zero"
else
   echo "You entered -ve number"
fi
```

Conditional Statement – if - else

```
echo "Enter body temperature in Farenheit"
read temp

if [ $temp -gt 99 ]
then
        echo "You got fever"
else
        echo "Normal temperature"
fi
```

Conditional Statement – if ...elif...else

```
echo "Enter 3 numbers "
read n1 n2 n3
if [ $n1 -gt $n2 -a $n1 -gt $n3 ]
then
        biggest=$n1
elif [ $n2 -gt $n3 ]
then
        biggest=$n2
else
        biggest=$n3
fi
echo "Biggest = $biggest"
```

```
#!/bin/bash
#conditional_statements_if_elif.sh

echo -n "Enter any number: "
read num

if [ $num -gt 0 ]
then
    echo "You entered +ve number"
elif [ $num -lt 0 ]
then
    echo "You entered -ve number"
elif [ $num -eq 0 ]
then
    echo "You entered zero"
else
    echo "Invalid Input" # Impossible Case
fi

echo "End of program"
```

Nested if

```
echo -n "Enter any number: "
read num

if [ $num -gt 0 ]
then
        echo "You entered +ve number ($num)"
else
        if [ $num -lt 0 ]
        then
                echo "You entered -ve number ($num)"
        else
                echo "You entered zero"
        fi
fi

echo "End of program"
```

Multiple Conditions – Case Statement

- The case statement begins with case and ends with esac. Each condition match is evaluated one after another. * matches with any value.

```
case "$1" in

        "-h") echo "Printing help"

        ...

        ;; # Break Statement

        ----

        *) echo "default"

esac
```

Decision Making – Case Statement

```
#!/bin/bash
#case_arith_ops_nums.sh
# Extend it for other arithmetic operations (*, /, %)

echo -n "Enter operand1 (number1): "
read num1
echo -n "Enter operand2 (number2): "
read num2
echo -n "Enter operation (1 for add or 2 for sub): "
read op

case $op in
1) res=`expr $num1 + $num2`
    echo "result addition: is $res"
    ;;
2) res=`expr $num1 - $num2`
    echo "result subtraction: is $res"
    ;;
*) echo "Unexpected input"
esac
```

```
#!/bin/bash
#case_arith_ops_symbols.sh
# Extend it for other arithmetic operations (*, /, %)

echo -n "Enter operand1 (number1): "
read num1
echo -n "Enter operand2 (number2): "
read num2
echo -n "Enter operation (+ for add or - for sub): "
read op

case $op in
+) res=`expr $num1 + $num2`
    echo "result addition: is $res"
    ;;
-) res=`expr $num1 - $num2`
    echo "result subtraction: is $res"
    ;;
*) echo "Unexpected input"
esac
```

```
#!/bin/bash
#case_vowel_or_consonant.sh

echo -n "Enter any character (a-z or A-Z): "
read mychar

case $mychar in
[aA]|[eE]|[iI]|[oO]|[uU])
        echo "You entered vowel"
        ;;
[b-d,B-D]|[f-h,F-H]|[j-n,J-N]|[p-t,P-T]|[v-z,V-Z])
        echo "You entered consonant"
        ;;
*)
        echo "You entered other than alphabet"
esac
```

case_execute_cmd.sh

```
#!/bin/bash
 echo –n "Enter a file name "
read filename
if [ ! -f $filename ]
then
       echo "Sorry, file does'nt exist“ ; exit
fi
echo "Enter cmd [cat/wc/sort/rm/ls] to run on the file : "
read cmd
case $cmd in
"cat"|"wc"|"sort") $cmd $filename
;;
"rm") $cmd -i $filename
;;
"ls") $cmd -l $filename
;;
*) echo "Error: cmd not in selection list"
;;
esac
```

Loops

- Loops enable you to execute a set of commands repeatedly .
  - while
  - for (for-in, usual for)
  - until

Loops

- while

```
while test expr
do
  Statement(s) to be executed if expr is true
done
```

```
num=1
while [ $num -le 10 ]
do
        echo -n "$num "
        num=`expr $num + 1`
done
echo
```

Iteration - while

- The while construct checks if a particular condition is valid or not, and executed the commands repeatedly as long as the condition is valid

- Example: Print the date 5 times

```
myval=0;

while [ $myval –lt 5 ]; do

        /usr/bin/date

        myval=`expr $myval + 1`

done
```

```
#!/bin/bash
#mult_table.sh

echo -n "Enter Table Number (to print multiple table): "
read tablenum

echo -n "Enter number of iterations: "
read iters

echo "Printing Table num $tablenum"
cntr=1
while [ $cntr -le $iters ]
do
        res=`expr $tablenum \* $cntr`
        echo "$tablenum * $cntr = $res"
        cntr=`expr $cntr + 1`
done
```

```
#!/bin/bash
#factorial_num.sh

echo -n "Enter a number: "

read num


fact=1

while [ $num -gt 1 ]

do

        fact=`expr $fact \* $num`

        num=`expr $num - 1`

done


echo "Factorial = $fact"
```

```
#!/bin/bash
#factorial_num.sh
echo -n "Enter a number: "
read num
if [ $num -le 0 ]; then
    echo "Invalid Input - Try again"; exit
fi
fact=1
cntr=1
while [ $cntr -le $num ]
do
        fact=`expr $fact \* $cntr`
        cntr=`expr $cntr + 1`
done
echo "Factorial of $num = $fact"
```

```
#!/bin/bash
#infinite_loop.sh
counter=1
while [ -1 ]
do
        echo "counter is $counter"
        counter=`expr $counter + 1`
        sleep 2
done
```

```
$ chmod u+x infinite_loop.sh
$ ./infinite_loop.sh
...
$ bg
....
$ fg
...
CTRL+C
$
```

Iteration - for

- The for loop executes for each value of the string that exists in a variable.
- Example

```
for val in $mylist
do
        echo "$val is the value"
done
```

Iteration - for

- Operates on lists of items. It repeats a set of commands for every item in a list.

```
for var in word1 word2 ... wordN
do
  Statement(s) to be executed for every word.
done
```

```
for str in "The Pursuit of happiness" Year 2006 Rating
7.9 IMDb
do
        echo $str
done
# Extend program to print number of sub strings
```

Loops - For

- Operates on lists of items. It repeats a set of commands for every item in a list.

```
for var in num1 num2 ... numN
do
  Statement(s) to be executed for every number.
done
```

```
for num in 1 5 25 10
do
        echo -n "$num  "
        sum=`expr $sum + $num`
done
echo
echo "Total sum of numbers is $sum"
```

```
#!/bin/bash
#for_loop_test.sh
for((cntr=1;cntr <= 20;cntr++))
do
        echo -n "$cntr  "
done
echo


for((cntr=2;cntr <= 10;cntr=cntr+2))
do
        echo -n "$cntr  "
done
echo
```

for_loop_ranges.sh

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

for num in {2..10..2}
do
        echo -n "$num  "
done
echo
```

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

for num in {10..1..-2}
do
        echo -n "$num  "
done
echo
```

```
#!/bin/bash

dirs=0
files=0
for myfile in `ls` ;do
        if [ -d $myfile ]; then
                dirs=`expr $dirs + 1`
        fi
        if [ -f $myfile ];  then
                files=`expr $files + 1`
        fi
done
echo "Total # of dirs= $dirs”
echo "Total # of regular files= $files”
```

Loops - until

```
until expr
do
  Statement(s) to be executed until expr is false
done
```

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

Loop Control Statements

- break
  - The break statement is used to exit the current loop (inner loop, if applicable)
- continue
  - The continue statement would ignore the current iteration and proceeds for the next iteration, if condition is success

```
#!/bin/bash
#  Sum numbers until 0
# sum_numbers_until_zero.sh
sum=0
while [ 1 ]
do
        echo -n "Enter number (0 to end): "
        read num
        if [ $num -eq 0 ]; then
                break
        fi
        sum=`expr $sum + $num`
done
echo "Sum of numbers is $sum"
```

break statement

```
#!/bin/bash
#  Sum numbers until 0 or maximum 3 numbers, which ever is first
sum=0
maxnums=3 # can change as required
cntr=0
echo -n "Enter number (0 to end): "
read num
while [ $num -ne 0 ]
do
   if [ $cntr -eq $maxnums ]; then
      break
   fi
   sum=`expr $sum + $num`
   cntr=`expr $cntr + 1`
   echo -n "Enter number (0 to end): "
   read num
done
echo "Sum of numbers is $sum"
```

sum_numbers_until_zero_or_max_cntr.sh

break statement

```
#!/bin/bash
sum=0

echo -n "Enter number (0 to end): "
read num
```

sum_pos_nums_until_zero.sh

```
while [ $num -ne 0 ]
do
  if [ $num -le 0 ]; then
    echo -n "Enter number (0 to end): "
    read num
    continue
  fi
```

continue statement

```
  sum=`expr $sum + $num`

  echo -n "Enter number (0 to end): "
  read num

done
echo "Sum of numbers is $sum"
```

```
#!/bin/bash

#  Sum numbers until 0 or maximum 3 numbers, which ever is first and ignore
   negative nums
sum=0
cntr=0
echo -n "Enter number (0 to end): "
read num
while [ $num -ne 0 ]
do
 if [ $num -le 0 ]; then
   echo -n "Enter number (0 to end): "
   read num
   continue
 fi
 sum=`expr $sum + $num`
 cntr=`expr $cntr + 1`
 echo -n "Enter number (0 to end): "
 read num
done
echo "Sum of numbers is $sum"
```

sum_pos_nums_till_zero_or_max_cntr.sh

continue statement

```
#!/bin/bash
#  Sum numbers until 0 or maximum 3 numbers, which ever is first and ignore negative nums
sum=0
maxnums=3 # can change as required
cntr=0
echo -n "Enter number (0 to end): "
read num
while [ $num -ne 0 ]
do
  if [ $num -le 0 ]; then
    echo -n "Enter number (0 to end): "
    read num
    continue
  fi
  sum=`expr $sum + $num`
  cntr=`expr $cntr + 1`

  if [ $cntr -eq $maxnums ]; then
    break
  fi
  echo -n "Enter number (0 to end): "
  read num
done
echo "Sum of numbers is $sum"
```

sum_pos_nums_till_zero_or_max_cntr.sh

break and continue statements

Nested Loops

```
#!/bin/bash
# mult_table_repeat.sh
echo -n "Enter Table Number (to print multiple table): "
read tablenum
echo -n "Enter number of iterations: "
read iters

while [ $tablenum -ne 0 ]
do
	cntr=1
	while [ $cntr -le $iters ]
	do
		res=`expr $tablenum \* $cntr`
		echo "$tablenum * $cntr = $res"
		cntr=`expr $cntr + 1`
	done
	echo -n "Enter Table Number (to print multiple table): "
	read tablenum
	if [ $tablenum -ne 0 ]
	then
		echo -n "Enter number of iterations: "
		read iters
	fi
done
```

Objectives

- Using arrays
- Functions
- Signal Handling
- Command Line Processing and Command Line Options
- Debugging
- Utilities (bc, cmp, diff, uniq, paste, join, cut, tr, sed)

Arrays

- Arrays provide a method of grouping a set of variables.

1. Assignment

   array_name[index]=value

2. Initialization

   array_name = (item1 item2 item3)

```
${arr[*]}                    # All of the items in the array
${!arr[*]}                   # All of the indexes in the array
${#arr[*]}                   # Number of items in the array
${#arr[0]}                   # Length of item zero
${arr[0]} or ${arr[1]}       # Referring an array element
```

```
#!/bin/bash
#array_test.sh

vehicle[0]="Bus"
vehicle[1]="Car"
vehicle[2]="Bike"
vehicle[3]="Cycle"
vehicle[4]="Cart"

echo "This is a  bullock ${vehicle[4]}"
echo "All vehicles: ${vehicle[*]}"
echo "Again all vehicles: ${vehicle[@]}"

echo "Indices of the vehicle array: ${!vehicle[*]}"
echo "Number of items in vehicle array: ${#vehicle[*]}"
echo "Length of item 4: ${#vehicle[3]}"
```

```
#!/bin/bash
#array_test.sh

vehicle[0]="Bus"
vehicle[1]="Car"
vehicle[2]="Bike"
vehicle[3]="Cycle"
vehicle[4]="Cart"

count=0
while [ $count -le 4 ]
do
  echo "Vehicle $count is ${vehicle[$count]}"
  count=`expr $count + 1`
done

len=${#vehicle[@]}
echo "Len of array is $len"
vehicle[$len]=123 #try with 123 "123" '123'
num=${vehicle[5]}
echo "num is $num"
num=`expr $num + 200`
echo "num is $num"
```

```
#!/bin/bash
# Summing of array elements, file name sum_array_elems_assg.sh
nums[0]=10
nums[1]=20
nums[2]=30

cntr=${#nums[*]}  # Number of array elements, in this case, 3
sum=0

while [ $cntr -gt 0 ]
do
  sum=`expr $sum + ${nums[$cntr-1]}`
  cntr=`expr $cntr - 1`
done

echo "Sum of array elements are $sum"
```

```
#!/bin/bash
fruits=(apple mango grapes)
echo All elements are
echo ${fruits[*]}

echo "${fruits[1]} is the king of fruits"
```

```
#!/bin/bash
#array_init.sh
fruits=(apple mango grapes banana orange)
len=${#fruits[@]}
echo "len is $len"
count=0
while [ $count -lt $len ]
do
        echo "fruit[$count] is ${fruits[count]}"
        count=`expr $count + 1`
        sleep 1
done
fruits[$len]="guava"
echo "${fruits[$len]}"

fruits[2]="avocado"
allfruits=${fruits[@]}
echo $allfruits
```

```
#!/bin/bash
# Summing of array elements, file name sum_array_elems_init.sh

nums=(10 20 30 40 50)

cntr=${#nums[*]} # Number of elements, in this case, 5

sum=0
while [ $cntr -gt 0 ]
do
  sum=`expr $sum + ${nums[$cntr-1]}`
  cntr=`expr $cntr - 1`
done
echo "Sum of array elements are $sum"
```

```
#!/bin/bash
# Summing of array elements using indices, file name
# sum_array_elems_init_indices.sh

nums=(10 20 30 40 50)
indices=${!nums[*]}
sum=0
for num in ${nums[*]}
do
  sum=`expr $sum + $num`
done
echo "Sum of array elements are $sum"

sum=0
for index in $indices
do
  sum=`expr $sum + ${nums[$index]}`
done
echo "Sum of array elements are $sum"
```

```
#!/bin/bash

nums=(10 20 30 40 [9]=50)

indices=${!nums[*]}
numelems=${#nums[*]}
items=${nums[*]}

echo "Indices: " $indices
echo "Number of elements: " $numelems
echo "All the items: " $items
```

```
#!/bin/bash
# array.sh
elems[0]=123
elems[1]='a'
elems[2]="SH"

echo "1. Elements are ${elems[*]}"
echo "2. Elements are ${elems[@]}"

elems[1]='S'
echo "3. Elements are ${elems[*]}"
echo "4. Elements are ${elems[@]}“

elems[3]="BASH"
echo "5. Elements are ${elems[@]}"

echo "Length of array is ${#elems[@]}"
echo "Length of element 2 is ${#elems[2]}"
len=${#elems[@]}
for ((count=0;count<$len;count++))
do
      echo "Element $count : ${elems[count]}"
done
```

Modularity - Subroutines

- At the beginning of the script, write all the subroutines

  mySubroutine() {

  }

- At any point later in the script, call the subroutine, with or without arguments

  mySubroutine hello world

- Subroutines extract the arguments just like the regular scripts would do ($1 etc)

- Subroutines can call other subroutines, provided they are already defined earlier in the script.

Functions

- Functions bring down the overall functionality of script into smaller subsections

- Syntax

```
function_name () {
    list of commands
}
```

```
Hello ()
{
  echo "Hello World"
}
# Invoke your function
Hello
Hello
```

Functions

```
# function-arg.sh
Hello ()
{
  echo "Hi, $1 $2"
}
# Invoke your function
Hello Lucky Me!
```

```
#!/bin/bash
# ip_noop.sh
fibo_series() # 1 1 2 3 5
{
    totalnums=$1
    num1=1
    num2=1
    echo "Fibonacci series are as follows:"
    echo -n "$num1  $num2"

    cntr=2
    while [ $cntr -lt $totalnums ]
    do
            sum=`expr $num1 + $num2`
            echo -n " $sum"
            num1=$num2
            num2=$sum
            cntr=`expr $cntr + 1`
    done
    echo
}
echo -n "Enter count to print fibonacci series: "
read cnt
fibo_series $cnt
```

```
#!/bin/bash
# noip_op.sh

# Function no input and output
myfunc()
{
    # Performing this since $?, can handle only values up to 255
    num=$((RANDOM%100))
    echo "In func: "$num
    return $num
}

myfunc
echo $?
```

```
#!/bin/bash

myfunc()
{
    local res=$1
    num=$RANDOM
    echo "In func: "$num
    eval $res=$num
}

myfunc randnum
echo $randnum
```

```
#!/bin/bash
# ip_op.sh
Sum ()
{
   num1=$1
   num2=$2
   res=`expr $num1 + $num2`
   return $res
}
# Invoke your function
Sum 17 6
echo Sum is $?

echo -n "Enter num1: "
read n1
echo -n "Enter num2: "
read n2

Sum $n1 $n2
echo Sum is $?
```

```
#!/bin/bash
# ip_op.sh
sum_numbers()
{
  cntr=$#
  sum=0
  for val in $@
  do
    sum=`expr $sum + $val`
  done
  #echo $sum
  return $sum
}

sum_numbers 10 20 30
echo $?

sum_numbers 10 20 30 40 50
res=$?
echo $res
```

```
#!/bin/bash
# Returning value more than 255

sum() {
   local res=$3
   tmp=`expr $1 + $2`

   eval $res=$tmp
}

sum 25000 5000 total

echo $total
```

```
#!/bin/bash
#func_sample.sh

source func_def.sh
# Invoke your function
Add1 10 20
Add2 20 30
echo "Main: Result is $?“

#!/bin/bash
#func_def.sh

Add1()
{
  res=`expr $1 + $2`
  echo "Add1: Result is $res"
}
Add2()
{
  res=`expr $1 + $2`
  return $res
}
```

Local and Global Variables

```
#!/bin/bash

print_local_global()
{
        local lvar="Linux"
        echo "in func: local variable is $lvar"

        gvar="UNIX OS"
        echo "in func: global variable is $gvar"
}

gvar="UNIX"
echo "in main: local variable is :$lvar:"
echo "in main: global variable is $gvar"
echo "calling function"
print_local_global
echo "in main: local variable is :$lvar:"
echo "in main: global variable is $gvar"
```

Executing other programs

- You can execute other programs/scripts by
  - Using absolute path
  - Using relative path from current directory
  - Depending on PATH environment variable
  - Preferred in some special cases but not recommended all the time
- Taking the output from a command
  - Keep it in an environment variable

```
DATE=`/usr/bin/date`
HOWMANY=`/usr/bin/who|/usr/bin/wc  -l `
```

  - Redirect the output to a file and process the file later

Including code from other scripts

- If you need some code to be used by several scripts, you can place it in a common file and include it in all the scripts

```
. mycommoncode.sh
```

Including code from other scripts

```
$ cat test_commoncode.sh
#!/bin/bash
. commoncode/mycommoncode.sh

mycommoncode
$
$ cat commoncode/mycommoncode.sh
mycommoncode()
{
        echo "In mycommoncode function"
}
$
$ ./test_commoncode.sh
In mycommoncode function
```

Input or Command Line Arguments

- The $ character is used to access input arguments to the script
  - $0 denotes name of the executing script
  - $1 points to first argument
  - $# gives the count of arguments
  - $n denotes nth argument
  - $* denotes all the input arguments
  - $@ denotes all the input arguments
  - $n denotes number of input arguments
  - $$ denotes PID of the executing script

```
#!/bin/bash
echo "Hello World from $0"
exit 0
```

```
#!/bin/bash
# Sum of command line arguments, up to 4 arguments
# cmdline_args_sum.sh
if [ $# -eq 4 ]
then
        sum=`expr $1 + $2 + $3 + $4`
elif [ $# -eq 3 ]
then
        sum=`expr $1 + $2 + $3`
elif [ $# -eq 2 ]
then
        sum=`expr $1 + $2`
elif [ $# -eq 1 ]
then
        sum=$1
else
        echo "You need to enter arguments from 1 to 4 only"
        exit
fi

if [ $# -ge 1 -a $# -le 4 ]
then
    echo "sum of numbers is $sum"
fi
```

```
#!/bin/bash
# Sum of command line arguments, if no arguments, sum until user enters 0
echo "Program to sum numbers as command line arguments or user input"
if [ $# -lt 1 ]
then
	echo "Syntax for command line args is $0 10 20 30"
	echo "Now proceeding with user input"
	echo -n "Enter Number (0 to end): "
	read num
	sum=0
	while [ $num -ne 0 ]
	do
		sum=`expr $sum + $num`
		echo -n "Enter Number (0 to end): "
		read num
	done
else
	sum=0
	while [ $# -gt 0 ]
	do
		num=$1
		sum=`expr $sum + $num`
		shift
	done
fi
```

Parameters processing

- shift
  - Shifts the script argument to left on every invoke of shift command

```
shift
```

```
echo "Argument List:  $*";
echo "Number of args: $#"
shift ; echo "Number of args: $#"
echo "Argument List:  $@";
shift ; echo "Number of args: $#"
shift ; echo "First arg is : $1"
```

```
$ ./shift_args.sh welcome to linux scripting
```

getopts

- This command is used to check valid command line argument are passed to script. Usually used in while loop.

```
getopts {optsring} {variable1}
```

- getopts is used by shell to parse command line options

- "optstring contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space.

- Each time it is invoked, getopts places the next option in the shell variable variable1, When an option requires an argument, getopts places that argument into the variable OPTARG. On errors getopts diagnostic messages are printed when illegal options or missing option arguments are encountered. If an illegal option is seen, getopts places ? into variable1."

```
#!/bin/bash
# Single Option
# getopts_single.sh
getopts aAbBcl opt
case "$opt" in
        a|A) echo "You entered a"
                ;;
        b|B) echo "You entered b"
                ;;
        c) echo "You entered c"
                ;;
        l) echo "You entered l"
                ;;
        *) echo "Invalid choice“
                ;;
esac
```

```
$ ./getopts_single.sh -a

$ ./getopts_single.sh -A –b

$ ./getopts_single.sh -b –A

$ ./getopts_single.sh -x
```

```
#!/bin/bash
# Multiple Options
# getopts_multiple.sh
while getopts aAbBcl opt
do
case "$opt" in
        a|A) echo "You entered a"
                ;;
        b|B) echo "You entered b"
                ;;
        c) echo "You entered c"
                ;;
        l) echo "You entered l"
                ;;
        *) echo "Invalid choice“
                ;;
esac
done
```

```
$ ./getopts_multiple.sh -a

$ ./getopts_multiple.sh -A –b

$ ./getopts_multiple.sh -b –A

$ ./getopts_multiple.sh –x

$ ./getopts_multiple.sh –a –b –c -A
```

```
#!/bin/bash
# Multiple Options with Value/s
# getopts_value.sh
while getopts a:b:cl opt
do
case "$opt" in
      a) echo "You entered a"
         avalue= "$OPTARG"
         echo "avalue is $avalue"
              ;;
      b) echo "You entered b"
                bvalue= "$OPTARG"
           echo "bvalue is $bvalue"
                ;;
      c) echo "You entered c"
              ;;
      l) echo "You entered l"
              ;;
      *) echo "Invalid choice"
              ;;
esac
done
```

```
$ ./getopts_value.sh –a 125

$ ./getopts_value.sh –c -a 125

$ ./getopts_value.sh –c –a 125 -l

$ ./getopts_value.sh –c –a 15 –l –b 25

$ ./getopts_value.sh –c –a 125 –l -b
```

Signal handling

- Unix signals (software interrupts) can be sent as asynchronous events to shell scripts, just as they can to any other program. The default behavior is to ignore some signals and immediately exit on others.
- Scripts may detect signals and divert control to a handler function or external program.
- This is often used to perform clean-up actions before exiting, or restart certain procedures.
- Execution resumes where it left off, if the signal handler returns.
- Signal traps must be set separately inside of shell functions.
- Signals can be sent to a process with certain keyboard combinations or with command kill.

- Following are pre-defined keys used to send signal to the current running process:
  - CONTROL+C → SIGINT → Terminate the process
  - CONTROL + Z → SIGTSTP → Stop the execution of current running process
  - CONTROL + \ → SIGQUIT → Terminate the process along with core dump
- In addition, CONTROL +D is not a signal, it's EOF (End-of-file) to end a program, it only works at the beginning of a line or if you do it twice (first time to flush, second time for read() to return zero)
- Can perform either of the following with the signal:
  - Default Action
  - Ignoring the signal
  - Handling/Catching the signal to override default action
- All signals except SIGKILL and SIGSTOP can be either ignored or handled
- You can use stty to check or change the characters that generate signals

```
$ stty -a |grep -Eow '(intr|quit|susp) = [^;]+'
```

USAGE: trap handler signal1,signal2…

- Handler is a command to be read (evaluated first) and executed on receipt of the specified sigs.
- Signals can be specified by name or number (see kill(1)) e.g. HUP, INT, QUIT, TERM etc, Ctrl-C at the terminal generates a INT.
- A handler of - resets the signals to their default values
- A handler of ‘ ‘(null) ignores the signals
- To get the list of signals issue command: $ kill -l

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).

Signal handling

- 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 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

- EXIT
  - the handler is called when the function exits, or when the whole
  - script exits. The exit signal has value 0.

```
$ kill ‘ ‘ pid #
```

```
$ kill -signal pid
```

```
trap huphandler  HUP
trap ''          QUIT
trap exithandler TERM INT

huphandler()
{
   echo 'Received SIGHUP'
   echo "continuing"
}

exithandler()
{
   echo 'Received SIGTERM or SIGINT'
   exit 1
}

seconds=0
while : ; do
# while true; do
    sleep 5
    seconds=$((seconds + 5))
    echo -n "$SECONDS $seconds - "
done
```

```
#!/bin/bash
trap myexithandler              SIGTERM
trap ''                         QUIT
trap myinthandler               INT
trap myusrhandler               SIGUSR1 USR2

myexithandler()
{
      echo "Received SIGTERM"
      exit 1
}
myinthandler()
{
      echo "Received SIGINT and now making to default"
      trap - INT # Check with CTRL+C
}
myusrhandler()
{
      echo "Received either SIGUSR1 or SIGUSR2"
}

while true ;
do
      echo "Welcome"
      sleep 3
      ps -f
done
```

Read File Contents Page wise

- Less - Read file page wise.
- More – Read file page wise.
  - Scroll backward (^b)
  - Scroll forward (Space Bar or ^f)
  - q for quit

```
$ cat generate_file.sh
#!/bin/bash
for line in {1..100}
do
    echo "This is Line $line" >> lines.txt
done
echo This is last line >> lines.txt
$ bash generate_file.txt
```

```
$ less lines.txt
```

```
$ more lines.txt
```

Shell Programming Summary

- You can use "bash –vx <script_name>" to run shell scripts in verbose mode
- Shell programming is very simple and powerful feature
- Shell scripts can be run on any Unix/Linux system
- Shell scripts are interpreted by the shell

debugging

- Set – useful for large scripts
  - -v, Starts debugging. Shows every statement. Substitutes values into variables
  - -x, Show statement execution status
  - +v, stop debugging.
  - +x, stop debugging.

Utilities

- bc
- cmp
- diff
- uniq
- paste
- join
- cut
- tr
- sed

bc

- bc - Basic Calculator

```
$ res=`echo 4.3+2.5|bc -l`
$ echo $res
6.8
$
```

cmp

- Compares two files byte by byte
- Displays the location of the first mismatch
- Syntax: $ cmp file1.txt file2.txt

```
$ cat file1.txt
This is text file
Last line of file1.txt
$ cat file2.txt
This is text file
Last line of file2.txt
$ cmp file1.txt file2.txt
file1.txt file2.txt differ: char 36, line 2
$
```

diff

- Displays file differences
- Syntax: $ diff file1.txt file2.txt

$ cat file1.txt

This is text file

Line 2 of file

Line 3 of file

Last line of file

$ cat file2.txt

This is text file

Line 2 of file

Last line of file

$ diff file1.txt file2.txt

$ diff file1.txt file2.txt

$ diff file1.txt file1.txt

uniq

- fetches one copy of each line
- Requires a sorted file as input
- Syntax: $ uniq file1.txt

```
$ cat > uniq_test.txt
this is line
this is line
file is uniq_test
this is line
end of file
$ uniq uniq_test.txt
this is line
file is uniq_test
this is line
end of file
$ uniq -u uniq_test.txt
file is uniq_test
this is line
end of file
$ uniq -c uniq_test.txt
      2 this is line
      1 file is uniq_test
      1 this is line
      1 end of file
$ uniq -d uniq_test.txt
this is line
$
```

```
$ cat uniq_test.txt
this is line
this is line
file is uniq_test
duplicate line
duplicate line
end of file
$ uniq -D uniq_test.txt
this is line
this is line
duplicate line
duplicate line
$ uniq -D --all-repeated=none uniq_test.txt  #Default
this is line
this is line
duplicate line
duplicate line
```

```
# Prepend empty line before each set of duplicate lines
$ uniq -D --all-repeated=prepend uniq_test.txt

this is line
this is line

duplicate line
duplicate line

# Prepend empty line between each set of duplicate lines
$ uniq -D --all-repeated=separate uniq_test.txt
this is line
this is line

duplicate line
duplicate line
$
$ uniq -u uniq_test.txt
file is uniq_test
end of file
```

paste

- Pastes vertically rather than horizontally
- Displays two or more files adjacently by pasting them
  - Syntax: $ paste file1.txt file2.txt
- Paste uses the tab as the default delimiter, we can specify one or more delimiter with –d option
  - Syntax: $ paste -d file1.txt file2.txt

```
$ cat states.txt
telangana
andhra pradesh
tamilnadu
kerala
karnataka
$ cat capitals.txt
hyderabad
amaravathi
chennai
thiruvananthapuram
bangalore

$ paste states.txt capitals.txt

$ paste -d ";" states.txt capitals.txt
```

Join

- A join of the two relations specified by the lines of file1 and file2
- Files are joined on a common key field (column) that should exist in both files.
- Both files must be sorted on the key field in the same order
- Join uses the tab as the default delimiter, we can specify one or more delimiter with –t option
  - Syntax: $ join -t"\t" file1.txt file2.txt

```
$ cat states.txt
1 telangana
2 andhra pradesh
3 tamilnadu
4 kerala
5 karnataka
$ cat capitals.txt
1 hyderabad
2 amaravathi
3 chennai
4 thiruvananthapuram
5 bangalore

$ join states.txt capitals.txt

# Ensure the file is separated with ; instead of default space or tab
$ join -t ";" states.txt capitals.txt
```

cut - Filter

- Cut - Cuts the character from the lines of file

```
$ cut -c 3-7,15-20 info.txt
```

```
$ cut -c 3-7,13,14,17-20 info.txt
```

```
$ cut -d":" -f1,6 /etc/passwd | tail -5
```

cut

```
$ cat lines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file
this is line 3 of file
this is line 4 of file
this is line 5 of file
this is last line
$
```

```
$ cut -c 14 lines.txt
$ cut -c 9-14 lines.txt
$ cut -c 1,9,14 lines.txt
$ cut -b 1,9-14 lines.txt
$ cut -b 14 lines.txt
$ cut -d: -f1,7 /etc/passwd|tail -5
```

tr

- Translates range of characters.
- *Syntax:*
  tr {pattern-1} {pattern-2}

```
$ tr "a-z " "A-Z"
hello
HELLO
Hi
HI
^d
$ echo "Hello world!" | tr "a-z" "A-Z"
HELLO WORLD!
$
```

```
$ cat tr_test.txt
welcome to
Linux basics and shell scripting
$
$ cat tr_test.txt |tr "[a-z]" "[A-Z]"
$ cat tr_test.txt |tr "[a-d]" "[A-D]"
$ cat tr_test.txt | tr [:lower:] [:upper:]

$ echo "Welcome to Linux"| tr [:space:] "\t"

$ echo "{Welcome to} Linux"| tr '{}''()'

$ echo "Welcome      to      Linux"| tr -s [:space:] ''

$ echo "Welcome to Linux"| tr -s [e] ''

$ echo "Welcome to Linux"| tr -d 'e'  # Delete specific character

$ echo "my ID is 40101716 at HCL"| tr -d [:digit:] # Deleting digit

$ echo "my ID is 40101716 at HCL"| tr -cd [:digit:] # Complement
```

sed

- SED is a stream editor.
- A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline).
- SED works by making only one pass over the input(s) and is consequently more efficient
- sed is used to edit (text transformation) on given stream i.e a file or may be input from a pipeline.
- *Syntax:*
  sed {expression} {file}

India's milk is good.

coffee Barista is good.

coffee is better than the tea.

Greetings of the day!

--Barista

```
$ sed '/coffee/s//milk/g' cafe.txt > milk.txt
```

```
$ cat sed_test.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix or linux which one you choose.
unix is easy to learn. unix is a multiuser os. Learn unix. unix is a powerful.

$ sed 's/unix/linux/' sed_test.txt
$ sed 's/unix/linux/2' sed_test.txt
$ sed 's/unix/linux/g' sed_test.txt
$ sed '3 s/unix/linux/' sed_test.txt # Replacing in 3rdline
$ sed 's/unix/linux/p' sed_test.txt # Print the matching line additionally

# Display only the matched lines, unmatched line not displayed
$ sed -n 's/unix/linux/p' sed_test.txt

# Replacing only from line 1 to line 3
$ sed '1, 3 s/unix/linux/' sed_test.txt
$ sed '2,$ s/unix/linux/' sed_test.txt
```

```
$ cat lines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file
this is line 3 of file
this is line 4 of file
this is line 5 of file
this is last line
$ sed '3d' lines.txt  # Deleting 3rdline
$ sed '$d' lines.txt  # Deleting lastline
$ sed '3,6d' lines.txt
$ sed '4,$d' lines.txt
$ sed '/line 2/d' lines.txt # Deleting the matching pattern line

$ sed G lines.txt # Inseting an empty line after each line
$ sed 'G;G'lines.txt # Inserting 2 empty lines after each line
$ sed '/^$/d;G'lines.txt # Removing empty lines and adding empty line after each line

$ sed '8i8 This is Line 8' lines.txt  # Inserting at line number 8
$ seq 3 | sed '2i 1.5'
```

```
$ cat lines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file

this is line 3 of file
this is line 4 of file
this is line 5 of file

this is last line

$ sed -i '3d' lines.txt # Effects the operations in the file itself
$ sed -i '6d' lines.txt

$ cat lines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file
this is line 3 of file
this is line 4 of file
this is line 5 of file
this is last line
$
```

sed

- Deleting all blank lines
  - As you know pattern /^$/, match blank line and d, command deletes the blank line.

```
    $ sed '/^$/d' mylines.txt
$ cat mylines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file

this is line 3 of file
this is line 4 of file
this is line 5 of file

this is last line
$ sed -i '/^$/d' mylines.txt
$ cat mylines.txt
this is line 1 of file again line 1 of file again again line 1 of file
this is line 2 of file
this is line 3 of file
this is line 4 of file
this is line 5 of file
this is last line
```