UNIT - 3
USING THE SHELL

- Using the Shell
- Command Line Structure
- Meta characters
- Creating New Commands
- Command Arguments and Parameters
- Program Output as Arguments
- Shell Variables
- More on I/O Redirection
- Looping in Shell Programs
- Important & Frequently asked questions

1.USING THE SHELL

- The Shell is a Program, that interprets user requests to run programs. The shell is an interface between the user and the kernel. The kernel does not know the human language ; hence shell accepts the commands from the user and converts them into a language that the kernel can understand.
- It is a program the user requests, calls programs from the memory and executes them one at a time.
- Several shells such as Bourne shell(sh), Korn shell(ksh), Bourne-Again shell(bash) and C(csh)shells are available.
- The shell also provide the facility of chaining or pipelining commands. This means that the output of one command is sent to the input of another command for further processing. In this manner ,one input data can be processed by several commands.
- There are TWO parts of a shell. The first is the **Interpreter**. The interpreter reads out commands and works with the kernel to execute them. The second part of the shell is **programming capability** that enables the user to write shell (command) script.
- A shell script is a file that contains a collection of shell commands to perform a specified task. It is also known as shell program.
- Shell is also called as a **processor** to process the user commands. Shell acts as a **command processo**r and **shell programming language**.

Functions of a shell:

- It parses the command line and identifies each and every word in it and remove additional spaces or tabs present if any.

- Evaluates all the variables and its data prefixed with $
- If commands are present within back quotes; they are executed and their output is substituted into the command line. i.e command substitution takes place.
- It then checks for any redirection of the input & output and establishes the connectivity between the concerned files accordingly.
- It then checks for the presence of the wild card characters lie *,?,&,[,]. if any of these characters present, file name generation & substitution takes place.

Features of Shell:

- Allows execution of commands and scripts
- Provides a set of built-in commands and utilities.
- Enables execution of commands in the background.
- Provides Input/Output redirection, pipes and filters.
- Enables command substitution by using back quotes
- Provides commands for loops and conditional branching.
- Supports pattern matching operators (?,*)

Writing shell scripts:

1.Comments should be preceded with a #.A comment split over multiple lines must have a # at the beginning of each line.

2.More than one assignment can be done in a single statement.

3.Multiplication symbol * must always preceded by a \ , otherwise the shell will treat it as a wild character. The wild character (*) usually represents any number of characters or numbers.

2.COMMAND LINE STRUCTURES

Command is just an instruction and it should be interpreted by the shell. The simplest command is a single word, usually naming a file for execution.

```
$ who          #  who command -  shows the users who are currently logged into a system
      guest-cse :0          2018-07-02 09:05 (:0)
      guest-cse pts/0       2018-07-02 10:48 (:0)
$
```

A command usually ends with a newline but a semicolon ; is also a command terminator:

```
$ who;
      guest-cse :0          2018-07-02 09:05 (:0)
```

guest-cse pts/0 2018-07-02 10:48 (:0)
$

Note: Both command structures will produce the same output.

$ who; ls; # unix shell executes **who** command first , after who it executes **ls** command.

guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)
Desktop Downloads file1 Music nfile Pictures rfile Videos
Documents examples.desktop lfile mydir1 ofile Public Templates
$

Similarly, unix runs **date** command first, later it can go for **who** command .
Here , **who** command doesn't run until **date** has finished.

$ date;who;

Mon Jul 2 10:49:23 IST 2018
guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)

Usage of pipe (|): Pipe is a unix utility that joins to utilities (or commands) by connecting the output of first command to the input of the second.

Example: Here , the output of **date** command is passed as the input of **wc** command. The wc command used to count the no.of Lines, Word & Characters in a file.
Ex:1
$ date | wc
First , the user must know the output of **date** command , later only the user can understand how the **pipe** will works.
$ date
Mon Jul 2 10:49:52 IST 2018

$ date | wc
1 6 29
Ex: 2
$date ; who | wc # here, pipe works on **who** command.
Mon Jul 2 10:50:56 IST 2018
2 10 60
In this example, after completion of **date** command only , shell can connects **who** command with **wc** command.

Connecting **who** and **wc** with a pipe forms a **single** command, called a pipeline , that runs after **date**.
Parentheses can be used to group commands.
Ex :1
$ (date;who)
Mon Jul 2 10:50:56 IST 2018
guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)

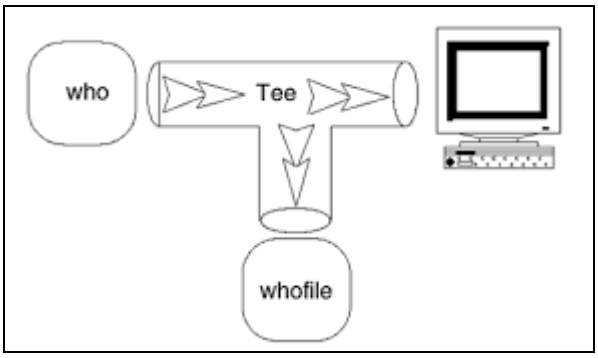
Ex: 2
$ (date;who) | wc

3 16 125

The outputs of date and who are concatenated into a single stream that can be sent down a pipe.

Unix also supports **data flowing** through a pipe can be tapped and placed in a file (but not another pipe) with a **tee** command.

Ex:



$ **(who) | tee whofile** # tee command shows the output of who command on a screen and sends this output to a file named "whofile"

guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)

To view the contents of a file, use cat command.

$cat whofile

guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)

Ex: 2

$ (date;who) | tee save |wc

3 16 125

$ cat save

Mon Jul 2 10:51:56 IST 2018
guest-cse :0 2018-07-02 09:05 (:0)
guest-cse pts/0 2018-07-02 10:48 (:0)

$ wc <save # here , wc command takes the input from save file.

3 16 125

Here, **tee** copies its input(date;who) to the named file as well as to its output , so **wc** receives the same data as if **tee** weren't in the pipeline.

The Unix allows another command terminator is the **ampersand (&)** . It is exactly like the semicolon or new line , except that it tells the shell not to wait for the command to complete. Typically, & is used to run a long-running command "in the background" while user can continue to type interactive commands:

$ long-running-command &

3432 #process id of long-running-command and then prompt appears immediately.
$
sleep command: the command that allows a script/command to wait for a specified time.
Syntax: sleep time_in_seconds
Ex: 1
$sleep 2 # 2 seconds pass before a prompt
$
Sleep command waits for the specified number of seconds before exiting:
Ex:2
$ (sleep 5; date) & date
3440 # process id
Mon Jul 2 11:01:26 IST 2018 # output from second date
$ Mon Jul 2 11:01:31 IST 2018 #prompt appears , then date 5 seconds later
Done (sleep 5; date)
Ex: 3

$ (sleep 5; echo tea is ready)& # tea will be ready in 5 minutes
3460 #process id
$ tea is ready #it will automatically displayed

Note: The & terminator applies to commands , since pipelines are commands you don't need parenthesis to run pipelines in the background.

pr command: It used to print the file.
Syntax: pr file_name
Ex: $pr file1
Ex: $pr file | lpr &

This command arranges to print the file on the line printer (lpr command) without making user wait for the command to finish. Parenthesizing the pipeline has the same effect, but requires more typing:
$ pr file1 | lpr & # without ( )
3472

$ (pr file1 | lpr) & # with( )
3472
Most programs accept arguments on the command line , such as file1 in the above example. Arguments are words , separated by blanks and tabs , that typically name files to be processed by the command.

Ex: ls -l file1 # to get the long list of file1.
The various special characters interpreted by the shell , such as < , > , | , ; and & are not arguments to the programs the shell runs. These are the special characters to control the shell to runs that commands.
Ex**:1**

$ echo Hello > file1 # the operator > for output redirection
$ cat file1
Hello
another structure to give the same output
$ **> file1 echo Hello**
$ cat file1
Hello
Ex :2
$ cat file1 | pr
2018-07-02 11:14 Page 1

Hello
$
$ pr <file1
2018-07-02 11:15 Page 1
Hello
$
$ pr file1
2018-07-02 11:13 file1 Page 1
Hello
Here, all three structures gives same output.

3.META CHARACTERS

- Meta characters are a group of characters that have special meanings to the UNIX operating system. Meta characters can make many tasks easier by allowing you to redirect information from one command to another or to a file, string multiple commands together on one line, or have other effects on the commands they are issued in.
- The shell recognizes a number of other characters as special; the most commonly used is the asterisk * which tells the shell to search the directory for filenames in which any string of characters occurs in the position of the * .
  Ex: echo *
- Special characters, or meta characters , have a special meaning to the shell. They can be used as wildcards to specify the name of a file without having to type out the file's full name. Some of the most commonly used meta characters are "*", "?", "[]", and "-".

The Asterisk(*) :

The * (asterisk) meta character is used to match any and all characters. Typing the following command will list all files in the working directory that begin with the letter l regardless of what characters come after it:

Ex: $ ls l*

The * (asterisk) meta character can be used anywhere in the filename. It does not necessarily have to be the last character.

The Question Mark(?) :

The ? (question mark) meta character is used to match a single character in a filename. Typing the following will list all of the files that start with "cse" and end with a single character :

Ex: $ ls cse?

Like the asterisk, the ? (question mark) meta character can be used as a substitution for any character in the filename.

Brackets ([ ]) :

Brackets ([…]) are used to match a set of specified characters. A comma separates each character within the set. Typing the following will list all files beginning with "a", "b", or "c":

Ex:$ ls [a,b,c]*

The Hyphen(-) :

Using the - (hyphen) meta character within [] (brackets) is used to match a specified range of characters. Typing the following will list all files beginning with a lowercase letter of the alphabet:

Ex: $ ls [a-z]*

If there are directory names meeting the specified range, their name and contents will also be displayed.

TIP: Although possible, it is highly recommended that you do not create file names containing meta characters.

>	Output redirection
>>	Output redirection (append)
<	Input redirection
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; one character
[ ]	File substitution wildcard; any character between brackets
`cmd`	Command Substitution
$(cmd)	Command Substitution
|	The Pipe (|)
;	Command sequence, Sequences of Commands
||	OR conditional execution
&&	AND conditional execution

( )	Group commands, Sequences of Commands
&	Run command in the background, Background Processes
#	Comment
$	Expand the value of a variable
\	Prevent or escape interpretation of the next character
<<	Input redirection . (Here Documents)

Table 3.1: Shell Metacharacters

`>`	*prog* `>`*file* direct standard output to *file*
`>>`	*prog* `>>`*file* append standard output to *file*
`<`	*prog* `<`*file* take standard input from *file*
`|`	p_1`|`p_2 connect standard output of p_1 to standard input of p_2
`<<`*str*	*here document*: standard input follows, up to next *str* on a line by itself
`*`	match any string of zero or more characters in filenames
`?`	match any single character in filenames
`[`*ccc*`]`	match any single character from *ccc* in filenames; ranges like `0-9` or `a-z` are legal
`;`	command terminator: p_1`;`p_2 does p_1, then p_2
`&`	like `;` but doesn't wait for p_1 to finish
`` `...` ``	run command(s) in ...; output replaces `` `...` ``
`(...)`	run command(s) in ... in a sub-shell
`{...}`	run command(s) in ... in current shell (rarely used)
`$1, $2` *etc.*	`$0...$9` replaced by arguments to shell file
`$`*var*	value of shell variable *var*
`${`*var*`}`	value of *var*; avoids confusion when concatenated with text; see also Table 5.3
`\`	`\`*c* take character *c* literally, `\`*newline* discarded
`'...'`	take ... literally
`"..."`	take ... literally after `$`, `` `...` `` and `\` interpreted
`#`	if # starts word, rest of line is a comment (not in 7th Ed.)
var`=`*value*	assign to variable *var*
p_1 `&&` p_2	run p_1; if successful, run p_2
p_1 `||` p_2	run p_1; if unsuccessful, run p_2

Shell interpretation :

echo :display a line of text. echo is often used to display or send an environment variable to a pipe.

syntax: echo [OPTION]... [STRING]...

OPTIONS

-n	do not output the trailing newline
-e	enable interpretation of the backslash-escaped characters listed below
-E	disable interpretation of those sequences in STRINGs
\b	for back space
\n	new line

\r carriage return
\t horizontal tab
\v vertical tab

How to avoid shell interpretation?

Sometimes we need to pass meta characters to the command being run and do not want the shell to interpret them. There are three options to avoid shell interpretation of meta characters.

1.Escape the meta character with a backslash (\).Escaping characters can be inconvenient to use when the command line contains several meta characters that need to be escaped.

2.Use single quotes (' ') around a string. Single quotes protect all characters except the backslash (\).

3.Use double quotes (" "). Double quotes protect all characters except the backslash (\), dollar sign (**$**) and grave accent (`).

Double quotes is often the easiest to use because we often want environment variables to be expanded.

Escaped characters:

\NNN	the character whose ASCII code is NNN (octal)
\ \	backslash
\a	alert (BEL)
\b	Backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

Here Documents (<<) :
This rather peculiar meta character (<<) allows us to provide complex, multi-line input to a command. Perhaps its most common usage is in a shell script used with the **cat** command to display a lengthy description to the user.
Ex:
$ cat << STOP
> Today, we hope that you learn a great deal about Unix.We are using Linux as our Unix system.
> There are many versions of Unix systems, including:UNIX, Solaris, BSD, AIX, HP-UX and Linux.
> References to UNIX, as it used to be spelled, are considered references to the UNIX from AT&T Bell Labs. >Unix is a generic term referring to all Unix like systems.

> STOP

output:

Today, we hope that you learn a great deal about Unix. We are using Linux as our Unix system.

There are many versions of Unix systems, including: UNIX, Solaris, BSD, AIX, HP-UX and Linux. References to UNIX, as it used to be spelled, are considered references to the UNIX from AT&T Bell Labs. Unix is a generic term referring to all Unix like systems.

In the above example, the > characters at the beginning of each line are produced by the **cat** command, which is reading from standard input. Those characters will not be present in shell script.

Note:

The Here Documents meta character produced the end-of-file character. It was not necessary to type Cntrl-d above.

In the above example, STOP is an identifier. Any string may be used as the terminal string.

File Redirection :

The Unix allows Standard Files and File Descriptors, standard input, standard output and standard error (stdin, stdout, stderr) are open file descriptors in every process.

1) stdin Redirection:

Redirect standard input from a file (instead of the keyboard) using the < metacharacter.

$ command < file

2) **stdout Redirection:**

Redirect standard output to a file (instead of the terminal) using the > metacharacter. The >>metacharacter redirects stdout appending the file.

$ command > file

$ command >> file

3) stderr Redirection:

Redirect standard error (error messages) to a file (instead of the terminal) using the 2> metacharacter. The 2>&1 metacharacter redirects stderr to the same file that stdout was redirected to.

stdout to terminal and stderr to file:

$ command 2> file

stdout and stderr to different files:

$ command > out_file 2> error_file

stdout and stderr to the same file:

$ command > file 2>&1

More examples with meta characters:

Ex: $ echo *

Desktop Documents Downloads examples.desktop file1 lfile Music mydir1 nfile ofile Pictures Public rfile save Templates Videos

Ex:$ echo .*

output:

. .. .bash_history .bash_logout .bashrc .cache .config .dbus .gconf .ICEauthority .kde .local .mozilla .profile .Xauthority .xsession-errors

Ex:$ echo '***'
output: ***

Ex:$ echo '***'
output: ***

Ex:$ echo ***
output: ***

Ex:$ echo "Don't do that!"
output: Don't do that!

Ex:$ echo x'*'y
output: x*y

Ex:$ echo '*'A'?'
output: *A?

Ex:$ echo 'hello > world'
output: hello > world

Ex:$ echo 'hello > world'

output: hello > world

Ex: $ echo 'hello
> world'
output:
hello
world

Ex: $ echo x*y
output: x*y

Ex: $ echo abc\

> def\

> ghi
output: abcdefghi

Ex: $ echo hello # there
output: hello

Ex: $ echo hello#there
output: hello#there

Ex: $ echo ls .*
output:
ls . .. .bash_history .bash_logout .bashrc .cache .config .dbus .gconf .ICEauthority .kde .local .mozilla .profile .Xauthority .xsession-errors

A digression on echo :
Unix allows **pure-echo** command to issue prompts from the shell.
Syntax: pure-echo prompt_message

Ex: $ pure-echo Enter a command:
output:
Enter a command : $ # No trailing newline

Drawback: Providing a newline –is not the default and takes extra typing:

Ex: $ pure-echo 'Hello!
> '
Output:
Hello!
$
Ex: echo –n Enter a command:
output: Enter a command: $ #prompt on the same line
Ex:$echo -
output: - # Only -n is special
$
Ex:$ echo \n '\n hello !\n how are
> you? `
output: hello !
how are you?

Ex:$ echo */* # lists all names in all directories
output:
Downloads/FS_UPDATED.doc mydir1/dir2
$

4. CREATING NEW COMMANDS

A command is a program that tells the Unix system to do something. It has the form.

Syntax: command [options] [arguments]

where an argument indicates on what the command is to perform its action, usually a file or series of files. An option modifies the command, changing the way it performs.

Commands are **case sensitive**.

Ex: command and Command are **not** the same.

Options are generally preceded by a **hyphen** (-), and for most commands, more than one option can be strung together, in the form:

syntax: command -[option][option][option]

Ex: ls -alR

will perform a long list on all files in the current directory and recursively perform the list through all sub-directories.

For most commands you can separate the options, preceding each with a hyphen

Syntax: command -option1 -option2 -option3

Ex: ls -a -l -R

Some commands have options that require parameters. Options requiring parameters are usually specified separately,

Ex:

lpr -P printer3 - #2 #file will send 2 copies of file to printer3.

How to create new commands in Unix shell?

Sometimes a sequence of commands needs to be repeated more than a few times . So, the user need to create new commands based on the old or existing commands.

Ex: To count the users frequently with the pipeline by use of a commands " $who |wc-l " . Instead of giving entire command , the user will create a new command i.e **nu** for listing no.of users.

Ex: $**who | wc**-l is same as $**nu** # counting **number of users** (**nu** command)only.

Step1: First now the users who are logged in by use of **who** command**.**

$who

x tty jul 20 9.00

y tty2 jul 20 9.30

z tty3 jul 20 9.45

step2: The output of who command will be given to word count with option l.

$who|wc - l

3

Step 3: The output of who|wc –l redirects to **nu** (new users) file / program / new command.

$echo 'who|wc – l' > nu

Here, nu – will gets input from terminal.

Step4: Display the contents of **nu** file using **cat** command.

$cat nu

who|wc-l

Step 5: Run this command through a shell. Here, shell gets input from nu only. (not from a terminal)

$sh < nu

3

Here, shell knows this is the small shell script. This command gives the output is same as the output from typed command at the terminal .i.e who|wc-l.

Step 6: Finally , use **nu** command for getting number of users.

$sh nu

3

Step7: Then the user need to make **nu** as **executable** file , so change its file permissions.

$chmod +x nu

Step8: Now the user can use **nu** command.

$nu

3

To make **nu** as the part of shell environment ,you have to fallow these steps.

Step1: use **pwd** command to know the path of a system

$pwd

/usr/cse

Step2: Create a **bin** directory by use of

$ mkdir bin

Step3 : Check once the **path** by use of

$echo $PATH

:/usr/cse/bin : /bin :/usr/bin

Step4: Install **nu**

$mv nu bin

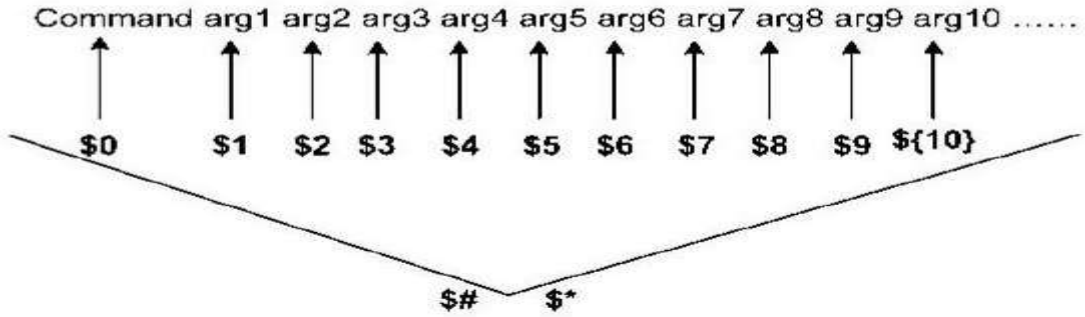
step5: use **nu** for number of users.

$nu

5. COMMAND LINE ARGUMENTS AND PARAMETERS

- The Unix shell is used to run commands and it allows users to pass runtime arguments to these commands.
- These arguments also known as "Command line parameters", that allows the users to either control the flow of the command or to specify the input data for the command.
- While running a command , the user can pass a variable number of parameters in the command line.
- Within the command script, the passed parameters are accessible using " positional parameters ".
- These range from $0 to $9, where $0 refers to the name of the command itself, and $1 to $9 are the first through to the ninth parameter , depending on how many parameters were actually passed.

- Command line arguments (also known as positional parameters) are the arguments specified at the command prompt with a command or script to be executed.
- The locations at the command prompt of the arguments as well as the location of the command, or the script itself, are stored in corresponding variables.
- These variables are special shell variables. Below picture will help you understand them.



Commonly used variables:

Variable	Description
$0	Represents the command or script.
$1 to $9	Represents arguments 1 through 9.
${10} and so on	Represents arguments 10 and further.
$#	Represents the total number of arguments.
$*	Represents all arguments.
$$	Represents the PID of a running script.

Unix / Linux - Special Variables

These variables are reserved for specific functions.

For example, the $ character represents the process ID number, or PID, of the current shell –

$1 - $9 These variables are the positional parameters.

$0 The name of the command currently being executed.

$# The number of positional arguments given to this invocation of the shell.

$? The exit status of the last command executed is given as a decimal string. When a command completes successfully, it returns the exit status of 0 (zero), otherwise it returns a non-zero exit status.

$$ The process number of this shell - useful for including in filenames, to make them unique.

$! The process id of the last command run in the background.

$- The current options supplied to this invocation of the shell.

$* A string containing all the arguments to the shell, starting at $1.

$@ same as $* , it represents list of separate strings.

Example:

Ex1. write a shell script to display all parameters.

```
$ cat greetings.sh
#greetings
#!/bin/sh
clear
hi.. goodmorning all...
echo " file name : $0"
echo " first parameter : $1"
echo  "second parameter : $2"
echo "third  parameter : $3"
echo " Total no.of arguments passed to the script  = $#"
echo "pid of the current file is = $$"
```

Run this file and pass the values for this script.

Syntax: $sh filename list_ of_ arguments

Ex:

```
$ sh  greetings.sh   how  are  you!
hi.. goodmorning all...
file name : greetings.sh
first parameter : how
second parameter : are
third  parameter : you!
Total no.of arguments passed to the script  = 3
pid of the current file is = 2364
```

Ex2: write a shell script to print positional parameters.

cat pp.sh

#positional parameters

#!/bin/sh

clear

echo “positional parameters”
echo ‘$0=’ $0
echo ‘$1=’$1
echo ‘$2’=$2
echo ‘$3=’ $3
echo ‘$4=’$4
echo ‘$5=’$5
echo ‘$6=’ $6
echo ‘$7=’$7
echo ‘$8=’$8
echo ‘$9=’$9
To run this , use $sh pp.sh arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9
$0=pp.sh
$1=arg1
$2=arg2
$3=arg3
$4=arg4
$5=arg5
$6=arg6
$7=arg7
$8=arg8
$9=arg9

Ex3:Write a shell script to accept some arguments and display it .

The commands echo $* and echo $@ both print the same thing, the list of all command line arguments, but “$*” is one string, and “$@” is a list of separate strings for each parameter. Consider the following two examples to see how these two similar variables differ.

a)cat myscript1

#sample script
#!/bin/sh
clear
for i in “$*”
do
	echo $i
done

output: ./myscript1 a b c

a b c

b) cat myscript2

```
#sample script
#!/bin/sh

clear
for i in "$@"
do
      echo $i
done
```

output: ./myscript2 a b c

```
a
b
c
```

Note: "$* "----- is one string and "$@ " --- is a list of separate strings for each parameter.

Ex4: Write a shell script to accept first and last name of a person

```
cat shellargs.sh
#first and last name of a person
#!/bin/bash
clear
echo "My first name is $1"
echo "My surname is $2"
echo "Total number of arguments is $#"
```

Output:

```
$ sh shellargs.sh  mani ch
My first name is mani
My surname is ch
Total number of arguments is 2
```

Ex 5. write a shell script to check a given number is even or odd.

cat ero.sh

```
#even or odd
#!/bin/sh
clear
echo "This script for finding a given number is even or odd"
if [ `expr $1 % 2` -eq 0 ]
then
       echo "the given number is even: $1 "
else
        echo "The given number is odd: $1"
 fi
```

output1:

$ sh ero.sh 25

This script for finding a given number is even or odd
The given number is odd: 25
output2:
$ sh ero.sh 20
This script for finding a given number is even or odd
the given number is even: 20

Ex6: write a shell script to perform all arithmetic operations without using command line arguments.

$cat > arth.sh

```
#arithmetic operations
#!/bin/sh
clear
a=40
b=40
val=`expr $a + $b`
        echo "a+b :" $val
val=`expr $a - $b`
       echo "a-b :" $val
val=`expr $a / $b`
       echo "a/b :" $val
val=`expr $a % $b`
       echo "a%b :" $val
val=`expr $a  \*  $b`
       echo "a*b :"  $val
if  [  $a  -eq  $b  ]
then
     echo "a is equal to b "
else
    echo " a is not equal to b "
fi
```

output:

```
$ sh arth.sh
a+b : 80
a-b : 0
a/b : 1
a%b : 0
a*b : 1600
a is equal to b
```

Ex6: write a shell script to perform all arithmetic operations t using command line arguments.

$cat > arthmetic.sh

```
#arithmetic operations using command line arguments
#!/bin/sh
```

```
clear
echo "Arithmetic Operations "
echo "add=`expr $1 + $2`"
echo "sub=`expr $1 - $2`"
echo "mul=`expr $1 \* $2`"
echo "div=`expr $1 / $2`"
echo "mod=`expr $1 % $2`"
```

$ sh arthmetic.sh 10 20

```
Arithmetic Operations:
add=30
sub=-10
mul=200
div=0
mod=10
```

Ex 7: Arithmetic with conditional expressions

```
$cat  add2.sh
 if  [ $#  -ge 2 ]
     then
         echo "add=`expr $1 + $2`"
         echo "sub=`expr $1 - $2`"
         echo "mul=`expr $1 \* $2`"
         echo "div=`expr $1 / $2`"
         echo "mod=`expr $1 % $2`"
else
    echo "  you have to provide two arguments"
fi
$ sh add2.sh 40 20
add=60
sub=20
mul=800
div=2
mod=0

$ sh add2.sh 10
  you have to provide two arguments

$ sh add2.sh
  you have to provide two arguments
```

We will now look at some additional commands to process these parameters**.**

1) SET COMMAND:

This command can be used to set the values of the positional parameters on the command line.

Example:

$set how do you do

$echo $1
how
$echo $2
do
$echo $1$2
how do
$echo $1 $2 $3 $4
how do you do

Here, “how” was assigned to $1 and “do” was assigned to $2 and so on.

2) SHIFT COMMAND:

This command is used to shift the position of the positional parameters. i.e. $2 will be shifted to $1 all the way to the tenth parameter being shifted to $9. Note that if in case there are more than 9 parameters, this mechanism can be used to read beyond the 9th.

All of the positional parameters $2 to $n to be renamed $1 to $(n-1), and $1 to be lost.

Syntax: shift no_of_postions

$ shift 2

Example:
$ set hello good morning how do you do welcome to Unix tutorial.
Here, ‘hello’ is assigned to $1, ‘good’ to $2 and so on to ‘to’ being assigned to $9.
Now the shift command can be used to shift the parameters ‘N’ places.
$shift 2
$echo $1
morning
$echo $2
how
$echo $9
tutorial

Now $1 will be ‘morning’ and so on to $8 being ‘unix’ and $9 being ‘tutorial’.

6. PROGRAM OUTPUT AS ARGUMENTS (WORKS LIKE COMMAND SUBSTITUTION)

Unix allows to pass the output of one program will be given as argument to other programs. The output of any program can be placed in a command line by enclosing the invocation in back quotes `.....`.

Ex1: echo Today's date is `date`
Today's date is Thu Jul 24 00:02:18 IST 2018
Here, date command is used as argument to echo command/program.
Ex2: echo "Today's
> date is `date` "
Today's
date is Thu Jul 24 00:02:18 IST 2018
$
Ex3: Suppose the user want to send a mail to a list of people whose login names are in the file **mailinglist**.

$mail `cat mailinglist` < letter

Here , **cat** command used to produce the list of user names , and those become the arguments to **mail** command.

Here , **mail** command is used to send emails to users .

Ex 4: The System User wanted to display their name.
$echo "Welcome to the system user $USER"
$Welcome to the system user Aliet

7.SHELL VARIABLES

Shell has variables to hold some values. Sometimes theses variables called as "parameters". Shell does not type-cast its variables .Variables can hold number or character data.

- A variable is a **location** in memory that is used to hold a value. This location is assigned a **name** to access the data. The value could be any type of data such as a name, number, text or filename/directory. So, a variable is nothing more than a pointer to a particular data.
- A shell allows a user to create, assign or delete variables. However, these variables are only **temporary** and are automatically deleted when the shell session is closed.
- To make a shell variable persistent and available system wide, it must be exported, thus converting it into an environment variable.
- The command used for doing this depends on the specific shell being used. In the Bash shell used by Linux, the command is "**export**".
- Variables can have alphabets, digits and underscore.
- Comma , white spaces are not allowed.
- Variable name must start with a letter or underscore
- variable names are case sensitive

A shell variable is created with a value. To assign a value , assignment operator(=) is used.

variable_name=variable_value
Ex: a=10
class=first
name=”Loyola engineering college”
id=16hp

For values with spaces, quotation marks must be used. Although not required, the convention in Unix is to use uppercase letters for the variable names. Also, in Unix, variable names, like filenames, are case sensitive.

Types of variables :

1)System defined/Predefined/ Unix/ Environment /Global variables
2)User defined/ Shell / Local variables

System defined variables: These variables are defined by the system , which have already special value. Also called as **Environmental** variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

Common Environmental variables:
Some environmental and shell variables are very useful and are referenced fairly often. Here are some common environmental variables that you will come across:

- **SHELL**: This describes the shell that will be interpreting any commands you type in. In most cases, this will be **bash** by default.
- **TERM**: This specifies the type of terminal to emulate when running the shell.
- **USER**: The current logged in user.
- **PWD**: The current working directory.
- **MAIL**: The path to the current user's mailbox.
- **PATH**: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **LANG**: The current language and localization settings, including character encoding.
- **HOME**: The current user's home directory.
- **HOSTNAME**: The hostname of the computer at this time.
- **IFS**: The internal field separator to separate input on the command line. By default, this is a space.
- **PS1**: The primary command prompt definition($). This is used to define what your prompt looks like when you start a shell session. The PS2 is used to declare secondary prompts (>) for when a command spans multiple lines.
- **UID**: The UID of the current user.
- **TZ**: Time Zone in which the user are working.

User defined variables:
These are defined by the user only. Also called as shell variables , which are available only to

the current shell. In contrast, an environment variable is available system wide and can be used by other applications on the system.

- Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.
- Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

A variable assignment is of the form *variable=value*, but its evaluation requires the $ as prefix to the variable name. Various examples ..

```
$count=5
$echo $count
$total=$count     #You can assign value of one variable to another variable
$echo  $total
```

Note: There should not be any space around =. i.e. if we say x =5 then the shell interprets x as command running with the =5 as argument!
All shell variables are of string type.
All shell variables are initialized to null strings by default.
i.e. x= [Enter] will assign null to x.
Note: For Constants in shell programming , **readonly** used

```
$age=30
$readonly  age
```

Use of Single & Double quotes:
When assigning character data containing spaces or special characters , the data must be enclosed in either single or double quotes.

- Using "double quotes" to show a string of characters will allow any variables in the quotes to be resolved.
  Ex: $var="text string"
  $var1="value of a variable is $var"
  $echo $var1
  output: value of a variable is text string
- Using 'Single quotes ' to show a string of characters will not allow variable resolution.
  Ex: $var='text string'
  $var1='value of a variable is $var'
  $echo $var1
  output: value of a variable is $var

Note: To list all environment variables , use "printenv " & To list all shell variables , use "set" command.
All environment variables are defined in /etc/profile , /etc/profile.d , ~/bash.profile. These files are the initialization files and they are read when the bash shell is invoked.

Unix also provides Positional variables($0,$1,$2.....$9) & Special variables ($*,$@, $$,$#,$?.. etc) to hold data when the user works through command line.

These parameters are comes under Environment variables.

Variable	Description
$0	Represents the command or script.
$1 to $9	Represents arguments 1 through 9.
${10} and so on	Represents arguments 10 and further.
$#	Represents the total number of arguments.
$*	Represents all arguments.
$$	Represents the PID of a running script.

System defined	User defined
1.Commonly called as Global shell variables	1.Commonly called as Local Shell variables
2.Contains system information or properties which system needs to run.	2.Contains users data only
3.Created and defined by the UNIX environment	3.Created and defined by the local user
4.These are used by the system and programs in the running of the overall of the system and accessed regularly. Such as required for each user profile. Such as your unique login id.	4.These are used by the user only. All local variables are applicable to current shell.
5.The **env (or) printenv** command list environment variables.	5.The **set** command list shell variables.
6.System administrator can change the system settings through environment variables	6.**Not possible** to change the system settings.
7.The variable name never changes and it is this name which is written into system configuration files and scripts.	7.The user itself can change or remove local variables and its data.
8.Written in upper case letters only.	8.Written in lower case letters only
9.These are available for all shells of all users.	9.A user can create their own local variables in any shell.But it will only be available in the current shell.
10.Ex: $PATH,$HOME,$PWD etc	10. Ex: name , age , marks etc.

8.STANDARD I/O STREAMS & REDIRECTION

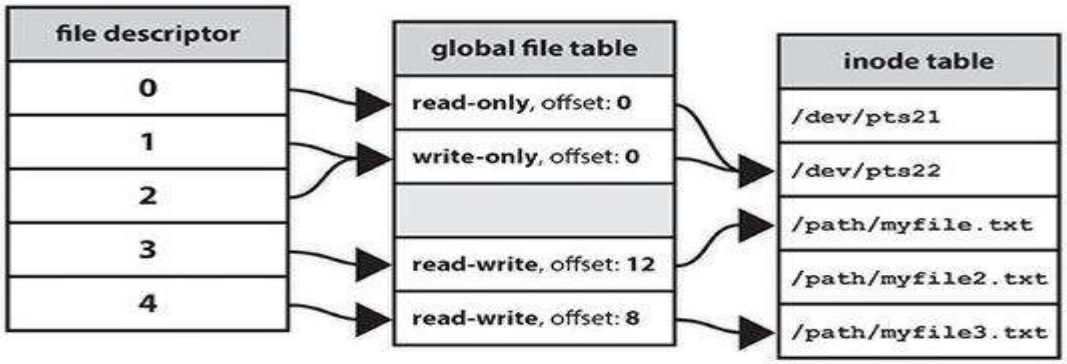
Unix supports an intersting mechanism called “Stream”. Streams are nothing but a sequence of bytes like FILE. There is no much difference between File and Stream. Unix allows THREE kinds of streams.

1.Standard Input
2.Standard Output
3.Standard Error

These Streams are nothing but Special files, that act as Communication Channels between an input source(such as File, Keyword, &Command) and a command. As well as a command and destination or target(such as File, Display Screen & command).These are called standard input, standard output & standard error. These communication channels are **special files** with file descriptors 0,1,2.

A file descriptor is a number that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

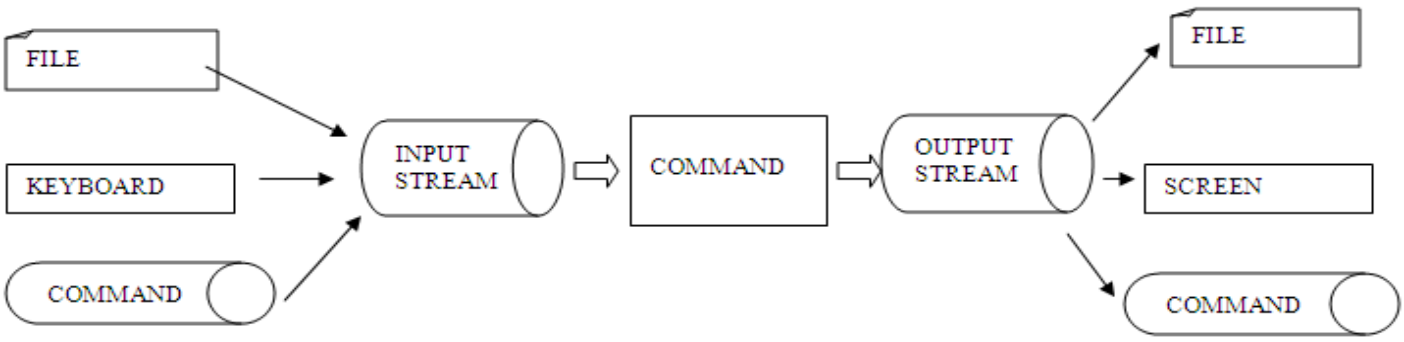
When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream (read-only, write-only, etc.).



Name	File descriptor	Description	Abbreviation
Standard input	0	The default data stream for input, for example in a command pipeline. In the terminal, this defaults to keyboard input from the user.	stdin
Standard output	1	The default data stream for output, for example when a command prints text. In the terminal, this defaults to the user's screen.	stdout
Standard error	2	The default data stream for output that relates to an error occurring. In the terminal, this defaults to the user's screen.	stderr

These THREE files together are known as “Standard I/O”.

Fig: Standard I/O.



Here, Command accepts input through input stream from input source(file, keyboard, command) and produces output on destination (file, screen, command) through output

stream .

The basic idea of standard I/O is to make every program is able to accept input from any source and write output to any target or destination.

Advantage: The user can design & develop programs without bothering about the variables in its input & output.
The Input may come from a disk file, the keyword or another program(command) through a pipe.
The output may go to any one of the destinations like a disk file, the display screen or another program(command) through a pipe.
When a command or program is initiated , the shell opens all the three standard I/O files automatically and attaches them to the command programs.
Bydefault, the shell makes the keyboard as the "standard Input" and the display screen as "Standard output".
The standard error also gets connected to the display screen by default.

Note: Not all commands use standard input & output. For example, Commands like rmdir, mkdir, cp, mv & others do not use standard I/O.

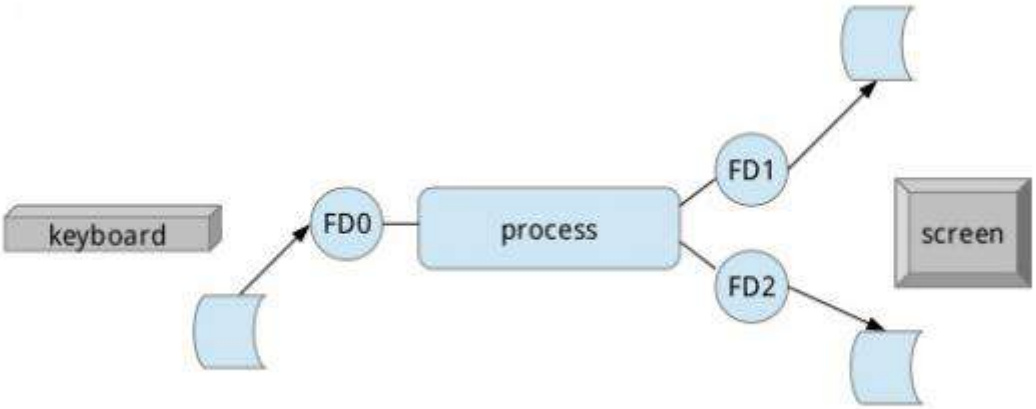
I/O Redirection:
A program or command takes its input via the standard input from any one source, the keyboard being default source, and directs its output via the standard output to any one destination, the screen being the default destination.
It is possible to change the source from where the input is taken by a program as well as the destination to where the output is sent by a program. This mechanism of changing the input source and /or output destination is called " Redirection"

In Unix, the Redirection is accomplished or done by using the following operators.
1)The Input redirection <
2)The Output redirection >
3)The output redirection with appending >>
4)The pipe connecting the output of one command as input to another command |

The input source is redirected (changed to otherthan the default source i.e keyboard) using the < (less than) operator .
The output destination is redirected using the > (greater than) operator or >> (double greater than) operators.
The file descriptors 0 & 1 are implicitly prefixed to the redirection operators < and > respectively by a shell.

The file descriptor 2 (that represents the standard error file) has to mentioned explicitly when required.

Note: The output of a program can be redirected using either > or >> operator.
When > is used , the destination files are overwritten.
When >> is used the present output will be appended to an existing file.
Note: In either case " is the destination file does not exist , it is created automatically.
Shell itself manages the all redirections .
Fig: IO redirection



Each process has 256 file descriptors. Processes use File descriptors to input or output (I/O) data with system. The first 3 file descriptors are standard
0: standard input
1: standard output
2:standard error

By default, each standard FD is connected to an IO device.
STDIN: keyboard
STDOUT: screen
STDERR: screen

Standard FD can be changed in command line: (when IO redirection done)
STDIN : <file
STDOUT : > file
STDERR : 2>file

Input Redirection: The input source is redirected (changed to other than the default source i.e keyboard) using the < (less than) operator .
Syntax:

command 0< file1 or ***command < file1***

Ex: **$wc < sample** #here the input has been redirected from a file called "sample"
3 20 103
Output Redirection: The output destination is redirected using the > (greater than) operator or >> (double greater than) operators.
Syntax:

command 1>	file1	or	command > file1
command 1>|	file1	or	command >| file1
command 1>>	file1	or	command >> file1

Ex: **$wc sample > newsample** # Here the output has been redirected to a file called "newsample"

Ex: It is possible to combine both < & > operators in a single command line.

$wc <sample > newsample

(or)

$wc >newsample <sample

(or)

$>newsample <sample wc

Here, < - will be processed first

>- will be processed next.

The < operator have a highest priority compared to the > operator.

Error Redirection:

Errors or messages can be stored exclusively in a file by redirecting them with the use of file descriptor 2 explicitly.

$cat sample 2> errorfile

or

$cat sample 2>> errorfile

The following table shows the redirection differences between the shells:

Type	Korn and Bash Shells	C Shell
Input	*0< file1* or *< file1*	< file1
Output	*1> file1* or *> file1* *1>| file1* or *>| file1* *1>> file1* or *>> file1*	*> file1* >| file1 >> file1
Error	*2> file2* *2>| file2* *2>> file2*	Not Supported Not Supported Not Supported
Output & Error (different files)	*1> file1 2> file2* *> file1 2> file2*	Not Supported Not Supported
Output & Error (same files)	*1> file1 2> & 1* *> file1 2> & 1* *1>| file1 2> & 1*	>& file1 >& file1 >&! file1

More Examples for Input , output & error redirection:

First , you need to create one text file named as **sample**

$cat > sample

This file for IO redirection. Various types are there. Such as Input redirection , Output redirection and Error redirection.

Ctrl+d

$ cat sample

This file for IO redirection. Various types are there. Such as Input redirection , Output

redirection and Error redirection.

Now, the data in "sample" file will be given as input to the **wc** command to count the no.of lines, words & characters.

$ wc < sample

2 20 40

Here, the input has been redirected from file called "sample". There is no display of the fiel name as the shell opens the file.

Output redirection: Here, the output has been redirected to another file, named as newsample.

Example:

$wc sample > newsample

$cat newsample

2 20 40

Example 2:

$ wc > myfile

This command can count for characters words and lines of a file and used for specific count

$ cat myfile

2 17 92

$ wc < myfile

1 3 24

$ cat myfile

2 17 92

$ wc > myfile

this is for word count of a file

$ cat myfile

1 8 33

$ wc < myfile > newfile # both input & output

$ cat myfile

1 8 33

$ cat newfile

1 3 24

$ wc > newsample < sample

$ cat newsample

12 12 204

$ cat sample

$ wc myfile > newsample

$ cat myfile

1 8 33

$ cat newsample

1 3 24 myfile

$ > newfile < myfile wc

$ cat newfile

1 3 24

$ cat myfile

1 8 33

```
$ cat myfile 2> errorfile
     1       8      33
$ cat newfile 2>> errorfile
 1  3 24
$ cat myfile 2>> errorfile
     1       8      33
$ cmp fil1 file2 > errorfile
cmp: fil1: No such file or directory
$ cmp file1 file2 > errorfile
$ cmp file2 file6 > errorfile
cmp: file6: No such file or directory
$ cat errorfile
$ wc errorfile
0 0 0 errorfile
$ wc
iam wc command
i count characters , words , and lines
with options i can also make selective count
     3      19     100
$ who > whoOct2
$ cat whoOct2
aliet    :0           2018-07-17 11:04 (:0)
aliet    pts/0        2018-07-17 15:40 (:0)
$ who > file1
$ cat file1
aliet    :0           2018-07-17 11:04 (:0)
aliet    pts/0        2018-07-17 15:40 (:0)
$ who > file1
$ who >| file1
$ cat file1
aliet    :0           2018-07-17 11:04 (:0)
aliet    pts/0        2018-07-17 15:40 (:0)
$ ls -l file1 nofile
ls: cannot access nofile: No such file or directory
-rw-rw-r-- 1 aliet aliet 88 Jul 17 16:43 file1
$ ls -l file1 nofile 1> filelist
ls: cannot access nofile: No such file or directory
$ more filelist
-rw-rw-r-- 1 aliet aliet 88 Jul 17 16:43 file1
$
$ more myerror
ls: cannot access nofile: No such file or directory
$ ls  file1 nofile 2> myerr 1> myout
$ ls -l file1 nofile 1> myout 2> myout
$ ls -l file1 nofile 1>| myout 2>| myout
$ more myout
-rw-rw-r-- 1 aliet aliet 88 Jul 17 16:43 file1
```

no clobber : To prevent unintentional clobbering (clobbering means overwriting happened when we used > operator) , use **set -o noclobber** command, it will prevent > from clobbering by making it issue an error message instead.
i.e clobbering a file is overwriting its contents .
Ex: $echo "Hello World" > file.txt
$cat file.txt
Hello World
$echo "This will overwrite the first greetings" > file.txt
$cat file.txt
This will overwrite the first greetings
This is the way to destroy the existing data in a file. To avoid this, use this command
$set -o noclobber

$echo " this is for testing " > file.txt
-bash:file.txt: cannot overwrite existing file.
If the user wanted to avoid noclobber function, use >| operator to ignore the noclobber.
Note: nocloober command always prevents redirected output from destroying an existing file.
To allow clobbering , use **$set +o noclobber.**

9. LOOPING (ITERATIVE / REPETITIVE)STRUCTURES IN SHELL

In Unix environment, to control or change the flow of execution of a program , we need various control structures.
Unix allows **three** types of **control structures**.

1) Sequential structures
2) Selection structures (Conditional structures)
3) Repetition structures (Looping or Iterative structures)

Sequential structures:
It means that , the program flows one line after another.

Normally, the shell processes the commands in a script **sequentially**, one after another in the order they are written in the file. Often, however, you will want to change the way that commands are processed.

Write a shell script to print hello world

$cat hai.sh

```
#!/bin/sh
clear
echo "Hello World"
```

output:

```
$sh hai.sh
Hello World
```

Write a script to add two numbers

$cat am.sh

```
#!/bin/sh
clear
a=10
b=20
val=`expr $a + $b`
echo "sum of a and b :" $val
```

output:

```
$ sh am.sh
sum of a and b : 30
```

Method 2:

$cat add2.sh

```
# take two integers from the user
#!/bin/sh
clear
echo "Enter two integers: "
read a b
sum=`expr $a + $b`
echo "Addition of two numbers : $sum"
```

$ sh add2.sh

```
Enter two integers:
30  20
Addition of two numbers :  50
```

You may want to choose to run one command or another, depending on the circumstances; or you may want to run a command more than once. To alter the normal sequential execution of commands, the shell offers a variety of control structures. Such as Conditional and Looping structures.

2) Selection structures / Conditional structures:

Code execution based on a logical decision. Unix shell supports conditional statements which are used to perform different actions based on different conditions.

There are **two** decision making statements:

1.The if..else statements

2 The case…esac statements.

1.The if… else statements:

If .. else statements are useful decision making statements , which can be used to select an option from a given set of options.

Unix shell supports the following forms of if..else statements.

1.1)if…then…fi statement
1.2)if…then…else…fi statement
1.3)Nested if..then..else…fi statement
1.4)if…then…elif…else...fi statement

1.1)The if…then..fi statement is the fundamental control statement , that allows shell to make decisions and execute statements conditionally.

Syntax:

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

Here, shell expression is executed. if the resulting value is **true**, given statement(s) are executed. If expression is **false** , then **no** statement would be not executed.

Example:

```
$cat  check.sh
#To Check whether the given numbers are equal or not
#!/bin/bash
clear
a=10
b=20
    if[ $a -eq $b ]
      then
          echo “a is equal to b”
     fi
   if [$a –ne  $b ]
      then
          echo “a is not equal to b”
    fi
```

Output: $sh check.sh

```
        a is not equal to b
```

1.2)The if…then…else..fi:

The next form of control statement that allows shell to execute statements in more controlled way and making decision between **two** choices.

Syntax:

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

fi

Example:

```
$cat  check.sh
#To Check whether the given numbers are equal or not
#!/bin/bash
clear
a=10
b=20
if [ $a –eq $b ]
  then
       echo " a is  equal to b"
  else
       echo "a is not equal to b"
fi
```

Output: $sh check.sh
a is not equal to b

1.3)The nested if..then..else..fi statement

The next form of control statement that allows shell to execute statements in more controlled way and making decision between **more than two** choices.

Syntax:

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

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

Example:

```
$cat  check.sh
#To find the maximum number of three numbers.
#!/bin/bash
clear
a=10
```

```
b=20
c=15
if [ $a -gt $b ]
  then
     if [ $a -gt $c ]
         then
              echo " a is  big"
      else
              echo " c is  big"
      fi
 else
    if[ $b –gt $c ]
       then
          echo " b is big"
    else
          echo "c is big"
    fi
fi
```

Output: $sh check.sh
b –is big

1.4)The if…then…elif…else...fi statement

It is the one level advance form of control statement that allows shell to make correct decision out of several conditions.

Syntax:

```
if [ expression 1 ]
   then
        Statement(s) to be executed if expression1  is true
elif[ expression 2]
    then
         Statement(s) to be executed if expression2  is true
elif[ expression 3]
    then
         Statement(s) to be executed if expression3  is true
    else
         Statement(s) to be executed  if  no expression  is true
fi
```

Example: Write a sell script that gives grades to student by specifying the marks between
80-100 give grade A
75-80 give grade B
60-75 give grade C
50-60 give grade D
40-50 give pass
below 40 give fail

```
$cat  grade.sh
#To find the grade of a student
#!/bin/bash
clear
echo “Enter marks”
read maks
if [ $marks  -ge 80  -a    $marks  -le  100]
   then
        echo “grade A”
elif [ $marks  -ge 75  -a    $marks  -lt  80]
   then
        echo “grade B”
elif [ $marks  -ge  60  -a    $marks  -lt  75]
   then
        echo “grade C”
elif [ $marks  -ge 50  -a    $marks  -lt  60]
   then
        echo “grade D”
elif [ $marks  -ge 40  -a    $marks  -lt  50]
   then
        echo “pass”
  else
       echo “fail”
fi
```

Output: $sh grade.sh

```
Enter marks
80
grade A
```

2 The case…esac statement

The user can use multiple if..elif statements to perform a multi-way branch. However , this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix shell supports **case ..esac** statement which handles exactly this situation , and it does so more efficiently than repeated if..elif statements.

Syntax:

```
case  word in
patten1) statement(s) to be executed if pattern1 matches   ;;
patten2) statement(s) to be executed if pattern1 matches   ;;
patten3) statement(s) to be executed if pattern1 matches   ;;
       :
       :
patten*)  default statement(s) to be executed if pattern does not matches ;;
esac
```

Example: Write a shell script it reads a digit from user and prints its textual spelling.

```
$cat print.sh
#prints  a number in textual form
#!/bin/sh

echo " Enter a number between 1 and 10. "
read num

case $num  in
        1) echo "one" ;;
        2) echo "two" ;;
        3) echo "three" ;;
        4) echo "four" ;;
        5) echo "five" ;;
        6) echo "six" ;;
        7) echo "seven" ;;
        8) echo "eight" ;;
        9) echo "nine" ;;
        10) echo "ten" ;;
        *) echo "Invalid number!" ;;
esac
```

Output:

```
$ sh print.sh
Enter a number between 1 and 10.
5
five
```

3) Repetition structures (Looping or Iterative structures)

A loop is an action or series of action repeated under control of loop criteria written by a programmer. Each loop tests a condition, if condition test is valid , then the loop continues until it becomes invalid. if the test is invalid , then the loop will be terminated.

In this, code execution is repeated based on a logical decision .Sometimes we want to run a command (or group of commands) over and over. This is called iteration, repetition, or looping. Unix allows various structures such as

1)**while** loop
2)**until** loop
3)**for** loop

while loop: It is used for repeating a set of statements for the time the specified logical expression is true.

The general form of the **while** loop is:

while condition
    do

```
        command(s)
    done
```

Examples:

Write a shell script to display a sequence of numbers from 1 to 10.

```
#!/bin/bash
n=1
while [ $n -le 10 ]
  do
     echo $n
     ((n++))                              #  it is  same as  n=’expr $n+1’
 done
```

output: 1 2 3 4 5 6 7 8 9 10

Write a shell script print the squares of integer from 1 to 5

```
#!/bin/bash
n=1
while [ $n -le 10 ]
  do
     expr $n  \*  $n
     n=’expr $n+1’
  done
```

output: 1 4 9 16 25

Write a shell script to print multiplication table.

```
#!/bin/bash
echo "Enter a Number"
read n
i=0
while [ $i -le 5 ]
do
   echo " $n x $i = `expr $n \* $i`"
   i=`expr $i + 1`
done
$ sh mt2.sh

Enter a Number
6
 6 x 0 = 0
 6 x 1 = 6
 6 x 2 = 12
 6 x 3 = 18
 6 x 4 = 24
 6 x 5 = 30
```

Write a shell script to find the sum of individual digits of a given number

```
#!/bin/bash
echo "enter the number"
read n
```

```
sum=0
while [ $n -gt 0 ]
 do
     temp=`expr $n % 10`
     sum=`expr $sum + $temp`
     n=`expr $n / 10`
 done
echo "sum of digits " $sum
```

Output:

```
enter the number
4562
sum of digits  17
```

Write a shell script to display factorial value of a given number

```
clear
echo "Enter n"
read n
f=1
while [ $n -gt 0 ]
do
f=`expr $f \* $n`
n=`expr $n - 1`
done
echo "Factorial is " $f
```

Output:

```
Enter n
5
120
```

Write a script for checking whether a given number is prime or not

```
#!/bin/bash
echo -n "Enter a number: "
read num
i=2
while [ $i -lt $num ]
do
  if [ `expr $num % $i` -eq 0 ]
  then
     echo "$num is not a prime number"
     exit
  fi
  i=`expr $i + 1`
done
echo "$num is a prime number "
```

output:

```
Enter a number: 13
13 is a prime number
$sh prime.sh
Enter a number: 10
10 is not a prime number
```

The until loop:

Another kind of iteration structure is the **until** loop. It is used for repeating a set of statements for the time the specified logical expression is false. The until command is the complement of the while loop , and its usage & form are similar.

while - repeat until the test condition is true.

until - repeat until the test condition is false

It has the general form:

```
  until condition
     do
        command(s)
     done
```

Write a shell script to display a sequence of numbers from 1 to 10.

```
#!/bin/bash
n=1
until [ $n -ge 10 ]
  do
     echo $n
     ((n++))                              #  it is  same as  n=’expr $n+1’
 done
```

output: 1 2 3 4 5 6 7 8 9 10

Write a shell script print the squares of integer from 1 to 5

```
#!/bin/bash
n=1
until [ $n -ge 10 ]
  do
     expr $n  \*  $n
     n=’expr $n+1’
  done
```

output: 1 4 9 16 25

Write a shell script to print multiplication table.

```
#!/bin/bash
echo "Enter a Number"
read n
i=0
until [ $i -ge 5 ]
do
```

```
    echo " $n x $i = `expr $n \* $i`"
    i=`expr $i + 1`
done
```

output:

```
Enter a Number
6
 6 x 0 = 0
 6 x 1 = 6
 6 x 2 = 12
 6 x 3 = 18
 6 x 4 = 24
 6 x 5 = 30
```

The for loop:

The most commonly used shell repetition structure is the **for** loop, which has the general form:

```
  for variable in list_of_values
    do
        command(s)
    done
```

Print the numbers from 1 to 5 using for loop

```
#!/bin/bash
for var in 1 2 3 4 5
  do
      echo $var
 done
```

output: 1 2 3 4 5

Write a script to print numbers in sequence from 5 to 10 using for loop

```
#!/bin/bash
for var in `seq 5 10`
  do
     echo $var
  done
```

output: 5 6 7 8 9 10

Write a script to print the number in reverse order using "rev" command

```
#!/bin/bash
clear
read -p "Enter a number: " num
echo $num | rev
```

output:

Enter a number: 564
465

Write a shell script to print multiplication table using for loop

```
#!/bin/bash
read n
for  i  in 1 2 3 4 5 6 7 8 9 10
  do
     echo  $i * $n =`expr $i \* $n`
 done
```

Output:

1 * 5 =5
2 * 5 =10
3 * 5 =15
4 * 5 =20
5 * 5 =25
6 * 5 =30
7 * 5 =35
8 * 5 =40
9 * 5 =45
10 * 5 =50

OPERATORS IN SHELL:

There are various operators supported by each shell. Based on default shell (Bourne), we are listing all Bourne Shell operators with examples..

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

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either awk or the must simpler program expr.

Here is simple example to add two numbers –

```
$cat   add.sh
#!/bin/sh
val=`expr 2 + 4`
echo “Total value : $val”
$sh add.sh
Total value: 6
```

There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.

Complete expression should be enclosed between ``, called inverted commas.

Arithmetic Operators

There are following arithmetic operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then −

Show Examples

Operator	Description	Example
+	Addition - Adds values on either side of the operator	`expr $a + $b` will

		give 30
-	Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
*	Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/	Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
=	Assignment - Assign right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.
All the arithmetical calculations are done using long integers.

Relational Operators:
Bourne Shell supports following relational operators which are specific to numeric values. These operators would not work for string values unless their value is numeric.
For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".
Assume variable a holds 10 and variable b holds 20 then −
Show Examples

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

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.
Boolean Operators

There are following boolean operators supported by Bourne Shell.
Assume variable a holds 10 and variable b holds 20 then −
Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true then condition would be true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators
There are following string operators supported by Bourne Shell.
Assume variable a holds "abc" and variable b holds "efg" then −
Show Examples

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

File Test Operators
There are following operators to test various properties associated with a Unix file.
Assume a variable file holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on −
Show Examples

Operator	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.	[-c $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-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.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes	[-k $file] is

	true.	false.
-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.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $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 execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.

Arguments or variables may be passed to a shell script. Simply list the arguments on the command line when running a shell script. In the shell script, $0 is the name of the command run (usually the name of the shell script file); $1 is the first argument, $2 is the second argument, $3 is the third argument, etc...
is a comment character
#! command specifies Command Interpreter

The special directive #! as the first characters of the file tell unix that the rest of the line identifies the program which should be used to run this file.

****

IMPORTANT & FREQUENTLY ASKED QUESTIONS

1) Explain different types of Meta characters with suitable examples

2) Discuss command line structures using shell

3) Explain How to create new commands in Unix

4) Explain the concept of command arguments and parameters

5) What is a shell variable? List and explain different types of variables in shell

6) What is IO Redirection ? Explain different types of redirections used in Unix

7) Briefly explain LOOPING in shell programs with examples

8) Write a shell script to display first n numbers of Fibonacci series.

9) Differentiate between while ,do-while and for loop with suitable examples

10) Explain shell responsibilities and explain the concept of "program output as arguments".