Module 2

File attributes and permissions

The shell’s interpretive cycle

Shell programming

Module 2

File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions.

The shell's interpretive cycle: Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. Connecting commands: Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions.

Shell programming: Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples.

File attributes and permissions

The ls command with options

-l option displays most attributes of a file like, *permissions, size* and *ownership*, which is referred to as **listing** in Unix language.

```
$ ls -l
total 120
drwxr-xr-x    4  kumar  staff   136  Apr   5  23:47  Applications
drwx------+  26  kumar  staff   884  Aug   9  14:33  Desktop
drwx------+   6  kumar  staff   204  Jun  14  20:59  Documents
drwx------+  11  kumar  staff   374  Aug   5  11:00  Downloads
drwx------@  73  kumar  staff   512  Apr  19  10:59  Library
-rw-r--r--@   1  kumar  staff  2445  Sep  14   2019  cq.c
-rw-r--r--    1  kumar  staff    92  Aug   5   2019  d.c
```

The list is preceded by the words **"total 120"**, which indicates that a total of 120 blocks are allocated by os to store information related to the files on pwd.

File attributes and permissions
The ls command with options

File Type and Permissions

First column shows *type* and *permissions* associated with each file.

First character in this column is mostly a -, indicating that the file is an ordinary file. If the first character reads *d,* it means it is a directory.

Next, 9 characters can be either r, w, x or -. In UNIX system, a file can have 3 types of permissions - *read*, *write* and *execute*.

Links

Second column indicates number of *links* associated with the file. This is actually the number of filenames maintained by the system of that file. A link count greater than one indicates that the file has more than one name(no two different copies of the file will be maintained).

File attributes and permissions
The ls command with options

Ownership

The user who has created the file will be the *owner* of the file.
Third, column displays *kumar* as the owner of all these files.
Owner has full authority to tamper with a file’s contents and permissions - a privilege not available with others except the root user.

Group Ownership

When a user is created, the system administrator assigns the user to some group.
Fourth column represents the *group owner* of the file.
Group privileges are generally distinct from others as well as the owner.

File attributes and permissions
The ls command with options

File Size

Fifth column displays the size of file in bytes (amount of data it contains).

It is only a character count of the file and not a measure of disk space it occupies.

Space occupied by a file on disk is usually larger than the number displayed, because files are written to disk in blocks of 1024 bytes or more.
Ex: d.c contains 92 bytes, which would occupy 1024 bytes on disk on system that use a block size of 1024 bytes.

Directory file size generally is 512 bytes, as directory maintains a list of filenames along with an identification number (the inode number) for each file. The size of the directory file depends on the size of this list.

```
$ df -k    // Displays, the amount of space allocated, and used for a directory.
```

File attributes and permissions
The ls command with options

Last Modification Time

Sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second.

A file is said to be modified only if its contents have changed in any way.

If permission or ownership of the file is changed, modification time remains unchanged.

If the file is less than a year old, since its last modification date, the year won't be displayed.

Filename

Last column displays filename arranged in ASCII collating sequence.

Ex: If a file name is chosen in uppercase - at least, its first letter, then it will be listed at the top of listing.

ls -l | grep '^d' lists only the directory names in the pwd.

File attributes and permissions

Changing File Permissions

When a file is created, username displays in the 3rd column of the file listing, user who has created the file is the **owner** of the file.

Group name is seen in 4th column, the group to which user belongs to is the **group owner** of the file.

If a file is copied from some other user, then the user who has copied the file will become the owner of it.

If a file is not able to be created in other user’s directory, it's because those directories are not owned by the current user.

Several users may belong to a single group.

The privileges of the group are set by the owner of the file and not by the group members

File attributes and permissions

Changing File Permissions

When system administrator creates a user account, the following parameters are assigned by him to the user.

The user-id (UID) - both its name and numeric representation

The group-id (GID) - both its name and numeric representation

/etc/passwd maintains UID (both number and name) and GID (but only the number).
/etc/group contains GID (both number and name).

```
$ id
uid=655537(kumar)   gid=655535(metal)
```

File permissions

```
-rwxr-xr--   1    kumar    metal …..
```

First column represents file permission. Unix follows 3-tiered file protection system that determines a file's access rights.

File attributes and permissions

File permissions

```
-rwxr-xr--   1     kumar     mithal …..
```

Initial -, in the first column represents an ordinary file.

```
      r  w  x        r  -  x     r  -  -
```

Each group here represents a category and contains 3 slots, representing **read, write** and **execute** permissions of the file.

r indicates read permission, which means **cat** can display the file.
w indicates write permission, meaning file can be edited.
x indicates execute permission; the file can be executed as a program.
- shows the absence of the corresponding permission.

First group (rwx) has all 3 permission, which is applicable to the **owner** of the file.
Second group (r-x) has read and execute permission for the **group member** of the file.
Third group (r--) has only read permission, applicable for **others**. Those who are neither the owner nor member of the group.

File attributes and permissions

File permissions

```
-rwxr-xr--   1     kumar     mithal …..
```

Even though kumar is a group member of mithal, group permissions are not applicable for kumar. Owner has its own set of permissions that override the group owner's permissions.

Chmod: Changing file permissions

When a file or directory is created a default set of permissions is automatically assigned to it by the system

```
$ cat > test
aaaaa^d^d
$ ls -l test
-rw-r--r--     1   kumar      metal      1906   Sep   5 23:38  test
```

As seen owner has got only r and w permission, where as group and others have got only read permission.

File/directory permission can be changed by the owner of the file using **chmod** command.

File attributes and permissions

Chmod: Changing file permissions

chmod command is used to set the permission of one or more files for all 3 categories of users i.e user, group and others.

chmod command can be executed only by the user/owner and the superuser.

chmod command can be used in two ways:

In a relative manner

In an absolute manner

Relative Permissions

Changing permissions in a relative manner, chmod only *changes the permissions specified in the command line* and *leaves the other permissions unchanged*.

Syntax: chmod *category* *operation* *permission* *filenames(s)*

File attributes and permissions

Relative Permissions

chmod takes as its argument an expression comprising letters and symbols that completely describe the user category and the type of permission being assigned or removed.

User *category* (user, group, others)
The *operation* to be performed (assign or remove a permission)
The type of *permission* (read, write, execute)

By using suitable abbreviations for each of these components, a compact expression can be framed and then use it as an argument to chmod.

Category	Operation	Permission
u - User g - Group o - Others a - All	+ Assigns permission - Removes permission = Assigns absolute permission	r - Read permission w - Write permission x - Execute permission

File attributes and permissions

Relative Permissions

To assign execute permission to owner/user of file test,

```
$ chmod  u+x  test
$ ls -l test
-rwxr--r--     1     kumar    ….   test
```

Command assigns execute permission to the user, but other permissions remain unchanged.

If test is an executable file it can be executed by the owner.

To enable others to execute the file

```
$ chmod   ugo+x    test   ;   ls  -l  test
```

String ugo combines all three categories - user, group and others. Synonym for ‘ugo’ is ‘a’. Other way round if no string is specified then permission is applied for all categories.

```
$ chmod  +x  test   ;  ls  -l  test
```

File attributes and permissions

Relative Permissions

chmod accepts multiple filenames. When same set of permissions are assigned to a set of files, this can be used.

```
$ chmod   +x   test    a.c    b.cpp
```

Permissions are removed with - operator. To remove read permission from both group and others

```
$ chmod   go-r   test   ;    ls  -l   test
```

chmod accepts multiple expressions delimited by commas.

```
$ chmod  a-x, go+r   test  ;     ls  -l  test
```

above command removes execute permission for all and adds read permission for group and others for the file test.

More than one permission can also be set for a file using chmod

```
$ chmod  u+rwx  test ;  ls  -l  test
```

File attributes and permissions
Absolute Permissions

In absolute mode, permission is set through octal numbers.

The only difference is that if absolute mode is used, it is necessary to represent permissions for each i.e u, g and o.

Octal numbers use the base 8, and octal digits have the values 0 to 7, which means that a set of 3 bits can represent one octal digit.

Binary	*Octal*	*Permissions*	*Significance*
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and Executable

Binary	*Octal*	*Permissions*	*Significance*
100	4	r--	Readable only
101	5	r-x	Readable and Executable
110	6	rw-	Readable and Writable
111	7	rwx	Readable, Writable and Executable

File attributes and permissions
Absolute Permissions

There are 3 categories of users and 3 permissions for each category, hence 3 octal digits can describe a file's permissions completely.

Most Significant Digit represents user and the least one represents others.

$ chmod   a+rw test   can also be entered as below

```
$ chmod 666  test ;   ls  -l  test
-rw-rw-rw- ………  test
```

```
$ chmod  761   test
```

Above command assigns all permissions to owner, read and write permissions to the groups and only execute permission to others.

File attributes and permissions

Recursively changing file permissions

It is possible to make chmod descend a directory hierarchy and apply the expression to every file and subdirectory, which is done by -R (recursive) option

```
$ chmod -R a+x  dir
```

Execute permission is applied to all files and subdirectories present within dir.

```
$ chmod 000 x.c
```

This permission renders the file virtually useless, file cannot be used for any purpose from *ugo*. sudo will override the restriction.

```
$ chmod 777 x.c
```

This type of command provides universal write permission. This type of file can be written by all. In order to achieve a secure system this type of permission must be avoided.

File attributes and permissions

Directory permissions

Directories also have their own permissions and the significance of these permissions differ from ordinary file.

Read and write access to an ordinary file are also influenced by the permissions of the directory.

```
$ mkdir  dir  ;   ls  -l | grep ‘^d’              //d option to display only directory
drwxr-xr-x      2     kumar     …..     dir
```

A directory must never be writeable by group and others.

The Shell’s Interpretive Cycle

Shell is the interface between user and the UNIX system.

Shell is a unique and multi-faceted program.

Shell is a combination of command interpreter and a programming language.

Shell is also a process that creates an environment for user to work in.

When user key-in, a command, it will be considered as input to the shell.
Shell first scans the command line for **metacharacters(** >, |, * etc).

Shell performs all actions represented by the symbol before the command can be executed.

```
$ rm -R *
```

shell replaces * with all file names in the pwd.

rm finally runs with these file names as arguments.

The Shell's Interpretive Cycle

When all pre-processing is complete, shell passes on the command to the kernel for final execution.

Command passed to kernel will not have any metacharacters that were originally used.

While the command is being executed, shell has to wait for notice of its termination from the kernel. After the command has completed its execution, shell once again issues the prompt to take up next command.

Summary of shell in it interpretive cycle

1. Shell issues prompt and waits for user to enter command
2. After a command is entered, shell scans the command line for metacharacters and expands abbreviations to recreate a simplified command line.
3. Shell then passes on the command line to the kernel for execution
4. Shell waits for the command to complete.
5. After command execution is complete, prompt reappears and the shell returns to its waiting state to start the next cycle, where in a new command can be entered.

Wild cards

$ ls chap*

Pattern chap* represents all filenames beginning with chap. This pattern is framed with ordinary characters and a metacharacter (*) following well defined rules (syntax).

The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

Wild Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character - either an i, j or k
[a-z]	A single character that is within the ASCII range of the characters a and z
[!ijk]	A single character that is not an i, j or k (Not in C Shell)

Wild Card	Matches
[!a-z]	A single character that is not within the ASCII range of the characters a and z (Not in C shell)
{pat1, pat2 …}	pat1, pat2, etc. (Not in Bourne Shell)

Wild cards

The * and ?

The metacharacter * matches any number of characters, including none.

$ ls chap*   lists all the files which starts with chap and has any characters, adjacent to it.
chap chap01   …. chapx   chapy   chapz.

chap* argument for ls matches chap also (i.e * matches none also).

When shell encounters this command, it identifies * immediately as a wild-card. It then searches in pwd and recreates the command containing those file names which start with chap.
Ex: ls chap chap01 chapx chapy chapz

Shell will handover the re-constructed command to kernel, which uses its process creation facilities to run the command

$ echo *   displays all the filenames in pwd.
$ rm *   command deletes all the files in pwd.

Wild cards

The * and ?

? wild-card matches a single character.

```
$ ls chap?
chapx  chapy   chapz
```

This command lists all the files starting with chap and any one single character after it. Consider, ? does not signify none like wild card *.

```
$ls chap??
chap01   chap02   chap03….
```

This command lists all the files starting with chap and any two single characters after it.

Matching with Dot

Hidden files in UNIX will start from '.' character. The * will not match all files beginning with a '.' or the / which is part of path.

```
$ ls  .???
```

lists all hidden file names having at least 3 characters after the dot.

Wild cards

'*' can be used to match any number of embedded dots.

(not the hidden file, which starts with dot)

Ex: the pattern apache*gz matches   apache_1.3.20.tar.gz.

* and ? don't match the / in a pathname.

Ex: cd /usr?local   cannot be used to match cd /usr/local

The Character class

Compared to * and ? more restrictive patterns can be generated with **character class**.

* and ? confines to more generic patterns

Character class contains a set of characters enclosed by the rectangular brackets [and], but it matches a single character in the class.

Wild cards

The Character class

Characters enclosed in square braces such as [abcd] is termed as a character class.
This matches a single character - an a, b, c or d. This can be combined with any string or another wild-card expression.

```
$ ls  chap0[124]
chap01   chap02  chap04
```

Range specification is also possible inside the class with a - (hyphen); the two characters on either side of it form the range of characters to be matched.

```
$ ls  chap0[1-4]              lists chap01, chap02, chap03 and chap04
$ ls  chap[x-z]               lists chapx, chapy and chapz
```

A valid range specification requires that the character on the left have a lower ASCII value, than the one on the right.

Note: Expression [a-zA-Z] matches all file names beginning with an alphabet, irrespective of case*

Wild cards

The Character class

Negating the Character Class (!)

The pattern which is part of a command can be negated using (!) symbol.

```
$ ls *.[!co]     lists all filenames with a single character extension but not the .c or .o files.
```

[!a-zA-Z]* matches all filenames that do not begin with an alphabetic character

Character class is to be used with a group of characters, and this is the only way to negate a match for a single character.

Matching Totally Dissimilar Patterns

To list all C and java source programs in pwd

```
$ ls   *.{c, java}
```

```
$ mkdir  ui  ;   cp *.{c,java}  ui
```

1st command creates a director ui and second command copies all .c and .java files from pwd to the newly created directory ui.

Wild cards

ls *.c   Lists all files with extension .c

mv * ../bin   Moves all files to bin subdirectory of parent directory

cp foo foo*   Copies foo to foo* (* loses meaning here).

cp ?????? progs   copies to progs directory all files with six character names.

lp note[0-1][0-9]   Prints files note00, note01 ….. Through note19.

rm * [!l][!o][!g]   Removes all files with three-character extensions except the ones with the .log extension

cp -r /home/kumar/{include, lib, bin}   test1
copies recursively the three directories, include, lib and bin from /home/kumar to the test1 directory.

Escaping and Quoting

File names in unix can be framed using metacharacters too (i.e *, ? etc.,)

Actual meaning of wild cards can be invalidated and can be considered as normal characters using two ways.

Escaping - providing a \ before wild-card removes its special meaning.

Quote - Enclosing wild-card, or even the entire pattern, within quotes like '*'.
Anything within these quotes are left alone by the shell and are not interpreted.

Escaping

```
$ touch  t1\*        create a file named t1*.    $ touch t1'*'
$ ls  -l  t1\*       long listing of the file name t1*      $ ls -l t1*'*'
$ touch  chap0[1-3]    creates a file named chap0[1-3]
Or
$ touch chap0\[1-3\]
```

Escaping and Quoting

Escaping the space

Apart from metacharacters, there are other characters that are special, like the *space character*.

The shell uses it to delimit command line arguments.
A filename or directory name can be framed by two words, having a space in between them.

```
$ touch My\ Doc ;  ls  -l My\ Doc
My Doc
$ rm My\ Doc
```

Escaping the \ Itself

In order to interpret \ itself literally, another \ must be appended before it.

```
$ echo \\
\
$ echo The newline character is \\n
The newline character is \n
```

Escaping and Quoting

Escaping the Newline Character

The newline character is also special delimiter, which marks the end of the command line.

Command lines use several arguments that can be long enough to overflow to the next line.

To ensure better readability, a single line can be split into two lines by keying in \ before enter and the command can be continued in the second line.

```
$ ls \
> My\ Doc
```

Output: My Doc

Command ls My\ Doc is split into two lines, by typing \ after ls and pressing enter, then the cursor moves to the second line and remaining part of the command can be entered, which will be considered by the Shell as a single command.

Escaping and Quoting
Quote
Another way to turn off the meaning of a metacharacter is to enclose metacharacters in quotes, the meanings of all enclosed special characters are turned off.
$ echo ‘\’
$ rm ‘chap*’ ; to remove only the file named as ‘chap*’ **$ rm chap’*’**
$ rm “My Document.doc”

Using escaping character ‘\’ becomes tedious when there are many wild cards to be nullified. Quote is often a better solution.

$ echo The characters |, <, > and $ are also special
Generates error

$ echo The characters \|, \<, \> and \$ are also special
Or by enclosing the string in single quote
$ echo ‘The characters |, <, > and $ are also special’

Escaping and Quoting - Quoting

Double quotes are more permissive/liberal. Double quotes considers $ and `(backtic) as interpretive characters and not as normal characters.

```
$ echo “Path value is = $PATH”   ;prints the PATH variable value
$ echo ‘Path value is = $PATH’      ; prints    Path value is = $PATH
```

Backtick is used within shell scripts. Backtick allows to assign the output of a shell command to a variable.

Single quotes won't interpolate anything, but double quotes will.

Ex: variables, backticks, certain \ escapes, etc.

```
$ cat > ms.sh
        #! /bin/bash
        testing=`date`
        echo “The date and time are: $testing“
        echo ‘The date and time are: $testing’^D
```

$ chmod u+x ms.sh

$ ./ms.sh

Escaping and Quoting - Quoting

```
$ echo “Command substitution used ` ` while SHELL is evaluated using $SHELL”
Output: Command substitution used   while SHELL is evaluated using /bin/bash
```

` ` is printing a space.

```
$ echo “List of files in pwd are `ls -l`, and shell used is $SHELL”
List of files in pwd are
X.c
T.c
…. /bin/bash
```

```
$ echo ‘yu file is created `cat > yu` and path is $PATH’   ;command cat > yu will not be executed
Output:  yu file is created `cat > yu` and path is $PATH
```

```
$ echo “yu file is created `cat > yu` and path is $PATH”
```

First “yu file is created” will not be displayed, if displayed the same must be copied to yu,
Hence, shell will execute cat > yu command first and the “yu file is created” will be printed, after this rest of the message will be printed.

Escaping and Quoting

Quoting

It is often crucial to select the right type of quote. Single quote protect all special characters.

Double quotes, lets a pair of backquotes be interpreted as *command substitution* characters, and the $ as a variable prefix.

There is also a reciprocal relationship between the two types of quotes; **double quotes protect single quotes and single quotes protect double quotes.**

```
$ echo “ ‘inside single quote’ “
‘inside single quote’
```

```
$ echo ‘ “inside double quote” ‘
“inside double quote”
```

Three standard files and redirection

"Terminal" is a generic name that represents the screen or keyboard.

Ex: Command output and error messages are displayed on the terminal i.e screen, users
provide command input through the terminal i.e keyboard.

Shell associates **3 files** with terminal - two for display and one for keyboard.

Shell associates these 3 files for all terminal related activities.

These special files are actually **streams** of characters which many commands see as input and output. A stream is a sequence of bytes.

When a user logs in, the shell makes available 3 files representing 3 streams. Each stream is associated with a default device.

Three standard files and redirection

Standard Input - File or stream representing input, which is connected to keyboard.

Standard Output - File or stream representing output, which is connect to display.

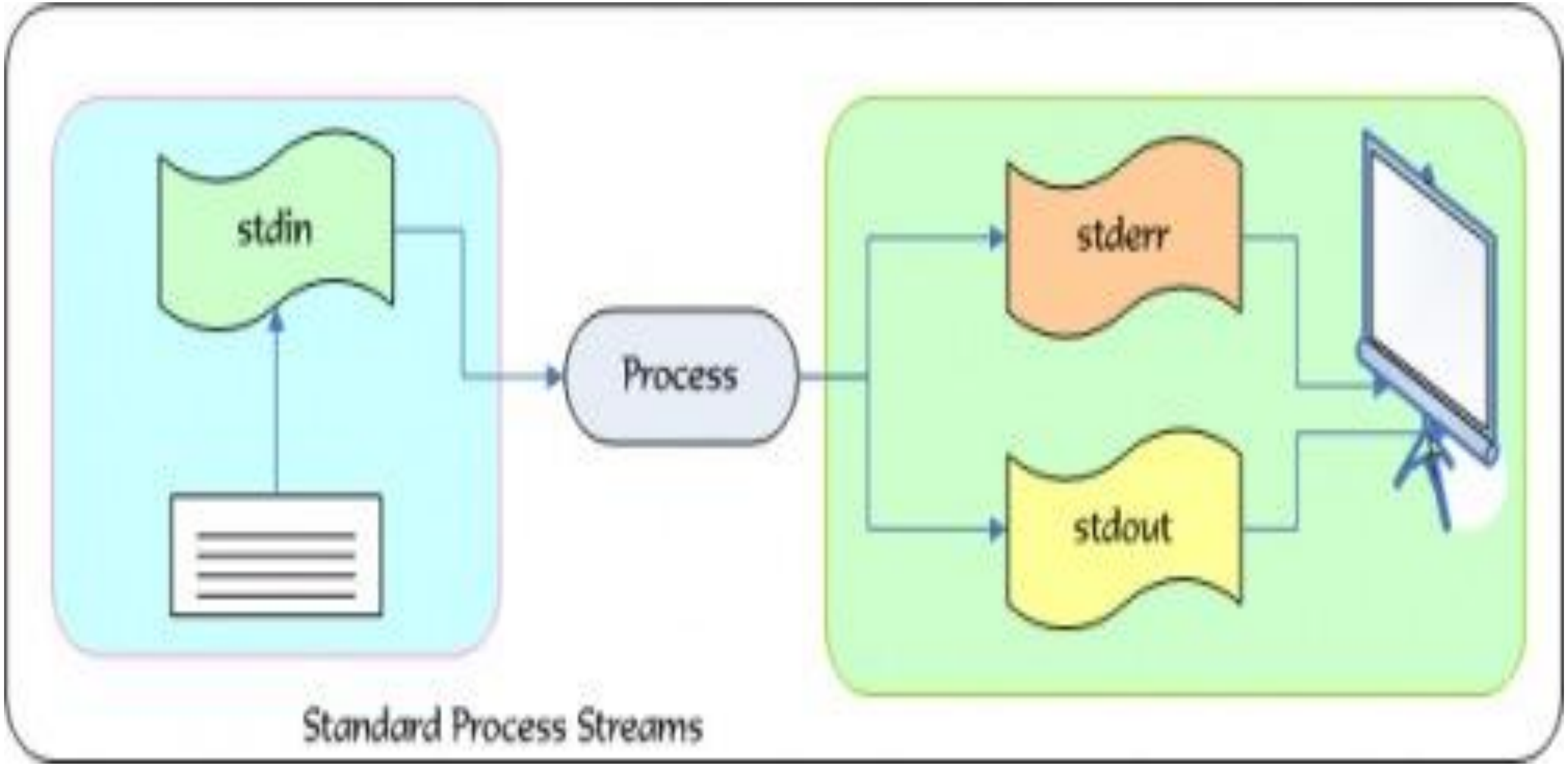
Standard Error - File or stream representing error messages that arise from the process or shell, which is also connected to display.

Every command that uses streams will always find these files open and available.

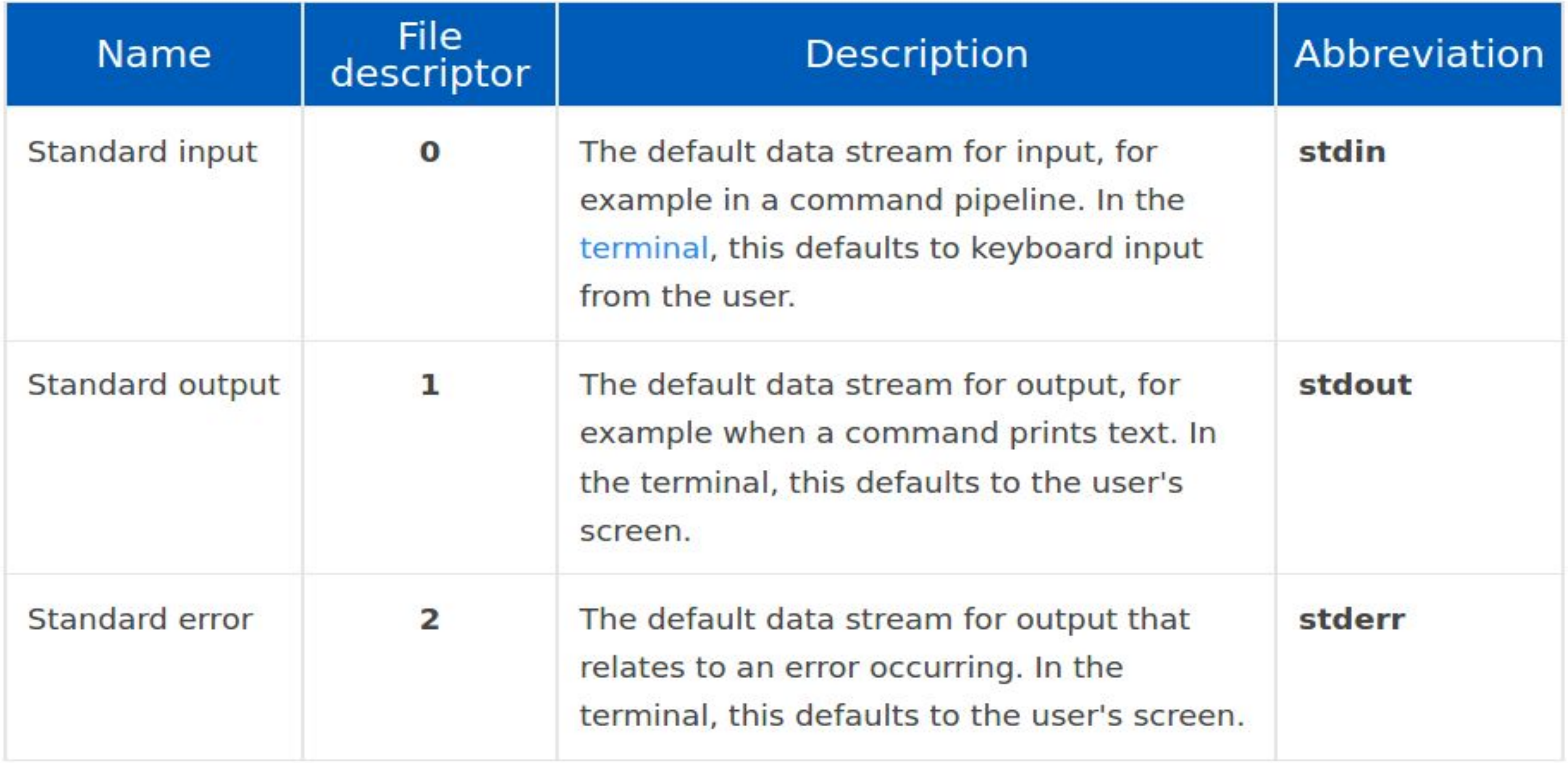
These files are closed when the command completes its execution.

Shell associates each of these files with a default physical device, where in this association is not permanent.

Shell will easily unhook a stream from its default device and connect it to a disk file (or to any command) the moment it seems special characters in the command line.
Ex: ls > t.txt


stdin
Process
stderr
stdout
Standard Process Streams

Three standard files and redirection
Standard Error



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**

Three standard files and redirection

Standard Input

Normal input for commands cat and wc are disk files.

When no disk file names are passed as an argument, to these commands, they read the file representing **standard input.**

Standard input file represents three input sources

Keyboard, the default source

A file using *redirection* with < symbol (a metacharacter)

Another program using a pipeline

Keyboard, the default source

When wc command is used without an argument and metacharacters in the command line, wc obtains its input from default source from keyboard and end of the input is marked by *[Ctrl-d]*.

```
$ wc
Standard input
Can be redirected [Ctrl-d]    1   5   32
```

// 1 line because ^d has been pressed in second line and there is no \n present in second line

Three standard files and redirection

Standard Input

Keyboard, the default source

wc, which takes the stream from standard input, immediately counts lines, words and characters.

wc when used with filename displays filename **($ wc cq.c)** as the fourth column output. When no filename is specified, filename has not been displayed in the fourth column.

In the previous example, **wc read a file i.e the standard input file.**

A file using redirection with the < symbol

Shell can reassign the standard input file to a disk file. It can **redirect** the standard input to originate from a disk. This reassignment or redirection requires the < symbol.

```
$ wc  <  x.c
3   14   71
```

Three standard files and redirection

Standard Input : *A file using redirection with the < symbol*

```
$ wc  <  x.c
3    14    71
```

File name will not be displayed, wc did not open x.c,
instead it read the standard input, from the file as a stream.

This redirection keeps the command ignorant of the source of its input.

```
$ wc  <  x.c
```

1. Upon receiving the <, shell opens the disk file, x.c for reading
2. It unplugs the standard input file as its default source and considers x.c as the standard input file
3. wc reads from standard input which has been reassigned by the shell to x.c

Three standard files and redirection

Standard Input

Taking Input Both from File and Standard Input

A Command can take input from a file and standard input both, by using - symbol .

$ cat - foo

Above command first accepts input from keyboard, which will be displayed first followed by the contents of file foo.

$ cat foo - bar

First contents of foo will will be printed, next input will be accepted from keyboard and displayed on screen finally contents of bar will be displayed on screen.

Three standard files and redirection

Standard Output

Commands displaying output on the terminal, writes to the **standard output** file as a stream of characters, and **not directly** to the terminal.

There are 3 possible destinations of this stream.

    The terminal, the default destination
    A file using the redirection symbols > and >>
    As input to another program using a pipeline

$ ls -l   displays file names with long listing option on the output terminal screen

$ wc sample.txt > newfile

1. Upon encountering >, shell opens the disk file, newfile, for writing.
2. Shell unplugs the standard output file from its default destination and assigns it to newfile.
3. wc writes to file newfile, which was reassigned by the shell (from standard output).

If ‘newfile’ does not exist it will be created first, if exists old contents of ‘newfile’ will be overwritten.

Three standard files and redirection
Standard Output

```
$ wc sample.txt >> newfile
```

Will append the output of the command wc to the newfile, if file already exists, its contents will not be deleted. If the file does not exist it will be first created.

```
$ cat *.c > progs_all.txt
```
copies all contents of .c file in pwd to the file progs_all.txt

Contents of the file copied to progs_all.txt will not be known the above command can be improvised as below.

```
$ ( ls -x *.c  ;  echo  ; cat *.c ) > progs_all.txt
```

echo command, inserts a blank line between the multicolumn file list and code listings.

Three standard files and redirection

Standard Error

Each of the 3 standard file is represented by a number, called a **file descriptor.**

A file is opened by referring to its pathname, subsequent read and write operations on file are identified by this file descriptor.

Kernel maintains a table of file descriptors for every process running in the system. First three slots are generally allocated to the three standard streams in this manner.

0-Standard input 1-Standard output 2-Standard error

These descriptors are implicitly prefixed to the redirection symbols.

Ex: > and 1> mean the same thing. `$ wc x.cpp 1> newfile`

< and 0< mean the same thing. `$ wc 0< x.cpp`

If a program in unix opens a file, the file will be having descriptor number 3.

Three standard files and redirection

Standard Error

```
$ cat  x.c                                        considering x.c file is not available
```

cat: cannot open a file

Command fails to open the file and writes to standard error. If the error message has to be redirected to a file, it has to be done as follows.

```
$ cat  x.c  2> err_file       just using > or >> will not move error message to a file.
```

```
$ cat >  x.txt ; cat  x.txt  > x1.txt  2> x2.txt
```

First command copies contents entered from keyboard to x.txt, next cat command will copy all its contents to x1.txt since file x.txt exists.

```
$ cat x3.txt > x1.txt  2> x2.txt
```

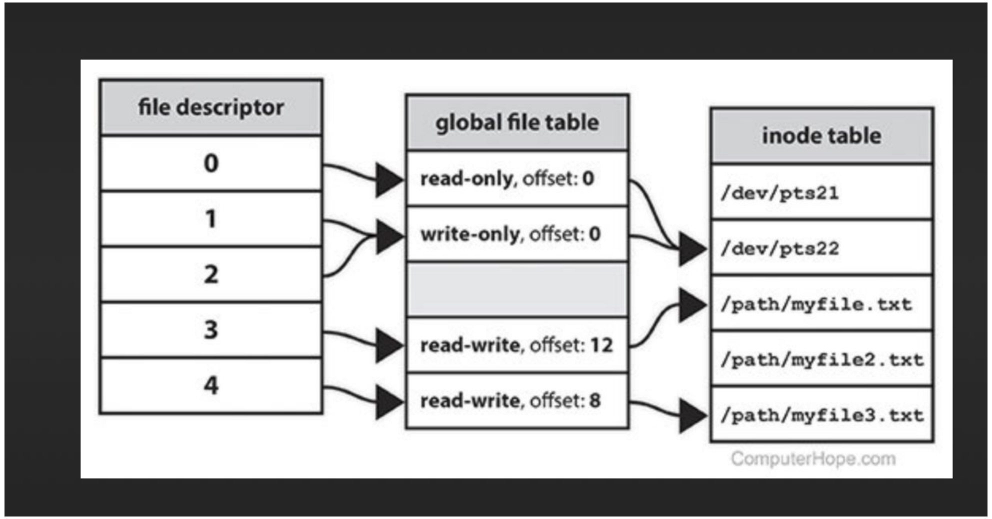
Assuming x3.txt file does not exist, error message will be copied to x2.txt, instead of displaying error message on standard display screen.

Three standard files and redirection : Standard Error

HOW REDIRECTION WORKS

Command like ls writes to file descriptor 1, which can be verified when the command is executed.

To save ls output in foo, the shell has to manipulate this file descriptor before running ls. It closes standard output and then opens foo. **Since, the kernel allocates the lowest unallocated integer in the file descriptor table, foo is assigned the value 1.** ls output is thus captured in foo.



Three standard files and redirection
Standard Error

This implementation requires two processes.

Shell creates a copy of its own process, performs the descriptor manipulation in the copied process and even executes ls command.

Shell’s own file descriptors are then left undisturbed(in parent process).

After completion of ls command the prompt will be displayed again, because the file descriptor 1 is already assigned to standard output file

Three standard files and redirection

Filters: *Using Both Standard Input and Standard Output*

Unix commands can be grouped into 4 categories based on the usage of standard input and standard output.

1. Directory-oriented commands like **mkdir**, **rmdir** and **cd** and basic file handling commands like **cp**, **mv** and **rm** use neither standard input nor standard output.

2. Commands like **ls**, **pwd**, **who** etc., don't read from standard input, but they write to standard output.

3. Command like **lp** that read standard input but will not write to standard output.

4. Commands like **cat, wc, od, cmp, gzip,** etc., that use both standard input and standard Output.

Commands in fourth category are called, in Unix parlance, **filters.** The dual stream handling feature makes filters powerful text manipulators. Most filters can also read directly from files whose names are provided as arguments.

Three standard files and redirection

Filters: *Using Both Standard Input and Standard Output*

```
$ cat > calc.txt
2^32
25*50
30*25 + 15^2
^D^D
```

bc is a command in unix which does the job of calculator.

```
$ bc < calc.txt  > result.txt
$ cat result.txt
```

bc command obtained the expression from redirected standard input, processed them and sent out the results to a redirected output stream. There is no restriction on the usage of < and > redirection operators.

Connecting Commands

Pipe

Pipe in Unix is used to pass the output of one process/command to another process/command.

`$ who > user.txt`	who command generates a list of users, one user per line, and the output of who is transferred to user.txt

Further, wc command can be used to count number of lines in user.txt, which is related to the number of users, using -l option

```
$ wc -l user.txt
```

Two commands are executed separately (who and wc), which has two disadvantages.

1. For long running commands, this process can be slow. Second command (i.e wc) cannot start execution unless the first has completed its job.

2. If the 1st command generates huge amounts of data, and if it is stored in a temporary file, it is wastage of disk space and it has to be removed after completion of 2nd command

Connecting Commands

Pipe

```
$ who | wc -l
```

Shell can connect streams with a special operator, the | (pipe), and avoid creation of the disk file. As seen in the command above both who and wc work in tandem so that one takes the input from the other.

Output of who is piped to wc. When multiple commands are connected this way a **pipeline** is said to be formed. It is the shell that sets up this connection and the commands have no knowledge of it.

Pipe is the source and destination of standard input and standard output respectively.

`$ ls | wc -l` counts the number of files in pwd, ls lists the files in pwd, one file in one line and wc counts the number of lines.

```
$ ls | wc -l > file_name.txt
```

Connecting Commands

Pipe

There is no restriction on the number of commands that can be used in pipeline.

```
$ cat  x.txt |  wc  |  cat > result.txt    or $ cat x.txt | wc > result.txt (without pipe)
```

x.txt contents will passed as an input to wc command and output of wc command is stored in result.txt

A pipe also has a built-in mechanism to control the flow of the stream.

Since, pipe is both read and written, the reader and writer has to act in unison. If one operator is faster than the other, then the appropriate driver has to readjust the flow.

```
$ ls | more
```

more freezes as long as user will not press a key, the kernel makes sure that ls writes to the pipe only as much as **more** can absorb at a time (i.e the length of the display unit).

Connecting Commands
Basic and Extended Regular Expression
grep: Searching for a pattern

Unix has a special group of commands for handling search requirements. grep command is the prominent member of this group.

grep scans its input pattern in a file and displays lines containing the pattern, the line number or filenames where the pattern occurs.

Syntax: *grep [options] pattern [filename(s)]*

grep searches for pattern in one or more filename(s) or the standard input if no filename is specified.

```
$grep "include" *.* | more
```

this command searches the pattern "include" in all the files in pwd and prints the result

Since, grep is a filter, it can search its standard input for the pattern and also save the standard output in a file

Connecting Commands

Basic and Extended Regular Expression : grep: Searching for a pattern

It is not necessary to enclose pattern in quotes (‘ ‘ or “ “), if more than one word is to be searched using grep command, then it has to be compulsorily enclosed in quotes.

When grep command is used with multiple filenames, it displays the file names along with the output.

```
$ grep “int main”  x.c    x2.c    x3.c
```

grep Options

Option	Significance
-i	Ignores case for matching
-v	Does not display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only

Option	Significance
-e exp	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line
-f file	Takes patterns from file, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings

Connecting Commands

grep Options

```
$ cat > emp.lst
1006|chanchal singhvi  |director |sales |03/09/38|6700
6521|lalit chowdary     |director |marketing |26/09/45|8200
9876|jai sharma            |director |production | 12/03/50|7000
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

Ignoring Case (-i)

The pattern specified with -i option, will search for the pattern, ignoring case.

```
$grep -i ‘agarwal’ emp.lst
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

Deleting Lines(-v)

grep’s -v option will select all lines except those containing the pattern

```
$ grep -v ‘director’ emp.lst
3564|Sudhir Agarwal   |executive | personnel | 06/07/47| 7500
```

Connecting Commands : **grep Options**

Displaying Line Numbers(-n)

-n option displays the line numbers containing the pattern along with the lines

```
$ grep  -n 'marketing'  emp.lst
2:   6521|lalit chowdary      |director |marketing |26/09/45|8200
```

Each output line is preceded by its relative line number in the file, starting at line 1.

Counting Lines Containing Pattern (-c)

-c pattern counts the number of lines containing the pattern (which is not the same as number of occurrences)

```
$ grep  -c 'marketing'  emp.lst
1
```

If this option is used with multiple files, the filename is prefixed to the line count.

```
$ grep  -c  'director'  emp*.lst
emp.lst:4
emp1.lst:2
```

Connecting Commands

grep Options

Displaying Filenames (-l)

-l option displays only the names of files containing the pattern

```
$ grep  -l  ‘manager’  *.lst
Desig.lst
emp.lst
emp1.lst
```

Matching Multiple Pattern (-e)

-e option can match any number of patterns

```
$ grep -e ‘dre’  -e ‘drec’   -e  ‘drect’   emp.lst
```

The lines in emp.lst matching at least anyone pattern in the pattern list will be listed.

Instead of having these lengthy expressions, having similar pattern, it can be replaced by **regular expressions** that generate patterns. By using regular expression a single expression can be used which generates patterns.

Regular Expressions - An Introduction

It is tedious to specify each pattern separately with -e option.

Like shell’s wild cards which match similar filenames with a single expression, grep uses an expression of a different type to match a group of similar patterns.

Unlike, wild cards, this expression is a feature of the ***command*** that uses it and has nothing to do with the shell.

Command uses an elaborate metacharacter set, overshadowing the shell’s wild-cards and can perform amazing matches.

Regular Expressions - An Introduction

Symbols or Expression	*Matches*
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, etc
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character within the ASCII range represented by c1 and c2
[1-3]	A digit between 1 and 3
[^pqr]	A single character which is not a p, q or r

Regular Expressions - An Introduction

Symbols or Expression	*Matches*
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line
bash$	Bash at end of line
^bash$	Bash as the only word in line
^$	Lines containing nothing

Regular Expressions - An Introduction

If a regular expression in the command uses any of the characters, listed in the previous table, it is termed as **regular expression.**

Regular expressions take care of some **common query** and **substitution requirements**.
Ex: To list similar names, in which required one can be selected. - Common query
To replace multiple spaces with a single space. - Substitution
To display lines that begin with a #.

Two categories of regular expressions exist - *basic* and *extended.*
Basic Regular Expressions (BRE) *Extended Regular Expression (ERE)*

Grep (file pattern searcher) command supports BRE by default and ERE with -E option.
sed (stream editor) command supports only BRE set.

The Character Class

A RE facilitates user to specify a group of characters enclosed within a pair of rectangular brackets [ ], in which case the match is performed for a *single* character in the group.

Regular Expressions

The Character Class

[ra] matches either ‘r’ or an ‘a’.

```
$ grep  [ra]  emp.lst      lists all the lines in emp.lst, because each line has a character
                           either ‘r’ or ‘a’
```

[aA]g[ar][ar]wal matches any names ‘agarwal’ or ‘<u>A</u>g<u>ar</u>wal’ or similar type of names. **A single pattern can match similar type of several names.** Character class [aA] matches the letter a in both lowercase and uppercase. Character model [ar][ar] matches any of the four patterns.

```
     aa      ar      ra      rr
```

```
$ grep “[aA]g[ar][ar]wal”  emp.lst     matches agarwal/Agrawal/agaawal/….
$ grep “[aA]g[ar][ar][wal]” emp.lst   will also list the above patterns
                                      [wal] searches for either w or a or l.
```

Pattern [a-zA-Z0-9] matches a single alphanumeric character, when range is used, character on the left of the hyphen must be a lower ASCII value than the one on the right. (Uppercase precedes lowercase in the ASCII sequence)

Regular Expressions

Negate Class (^)

RE use ^ to negate the character class, when a character class begins with this character, all characters other than the ones grouped in the class, are matched.
Ex: [^a-zA-Z] matches a single non-alphabetic character string.

NOTE: -v option is better than ^ for nullifying more than 1 character in the pattern.

The *

* (asterisk) refers to the *immediately preceding* character. In grep command it indicates that the preceding character can occur many times, or not at all.
g*
Matches single character g, or any number of g's. Since, there is a possibility that the previous character (i.e g in example) may not occur at all, it also matches a null string. Hence, apart from null string, it also matches following strings: g gg ggg gggg ……

"Zero or more occurences of the previous character"

In order to match a string beginning with g; gg* has to be used.

Regular Expressions

The *

```
$ grep "[aA]gg*[ar][ar]wal" emp.lst
```

The pattern matches “aggarwal”, “Agarwal” and “agrawal” all three similar names but with different spellings. RE, in command is not limited to match only 3 names.

The Dot

A ‘.’ matches any single character.

Ex: 2…   matches four character pattern beginning with a 2

RE .*, signifies any number of characters or none. In order to find a name that starts with ‘j’, the pattern will be

```
$ grep “j.*” emp.lst
```

RE to find “J.B. saxena” or “J.B. agarwal”

```
$ grep “J\.B\..*”  emp.lst
```

Actual meaning of dot is escaped with \, which is used for despecializing next character.

Regular Expressions

Specifying Pattern Locations (^ and $)

Most of RE characters are used for matching patterns, but ^ and $ are used to match a pattern at the beginning or end of a line.

^ - For matching at the beginning of a line $ - For matching at the end of a line.

$ grep “^2” emp.lst to list those employees, whose employee number starts with 2.

$ grep “7…$” emp.lst

Lists those employees, whose salary lies between 7000 to 7999. ‘$’ will force grep command to search at the end.

$ grep “^[^2]” emp.lst Selects only those line whose emp-id’s do not begin with a 2.

$ ls -l | grep “^d” to list only directories in pwd

Regular Expressions
ERE and grep
ERE makes it possible to match dissimilar patterns with a single expression.

Expression	**Significance**
ch+	Matches one or more occurrences of character *ch*
ch?	Matches zero or one occurrence of character *ch*
exp1 | exp2	Matches exp1 or exp2
GIF | JPEG	Matches GIF or JPEG
(x1 | x2) x3	Matches x1x3 or x2x3
(lock/ver) wood	Matches lockwood or verwood

Regular Expressions
ERE and grep
ERE expressions are executed with -E option

The + and ?
ERE set includes 2 special characters + and ?. They are often used in place of * to restrict the matching scope.

+	Matches **one or more** occurrences of the previous character
?	Matches **zero or one** occurrence of the previous character

Ex: b+ match b, bb, bbb etc., The expression b? Matches either a single instance of b or nothing.

```
$ grep -E  “[aA]gg?arwal”  emp.lst
```
This command is used to find employee names like *“anil* ***aggarwal****”* or *“sudhir* ***Agarwal****”*

Regular Expressions

The + and ?

To list #include <stdio.h>, #include   <stdio.h>, #include   <stdio.h>, which is a multi word string. Considering there is no space between ‘#i’ and there may be more than one space between ‘include’ and ‘<stdio.h>’.

```
$ grep -E -n “# ?include +<stdio.h>”  *    //-n additionally displays line number with file name.
```

Matching Multiple Patterns (|, (and))

The | is the delimiter of multiple patterns.

```
$ grep -E ‘sengupta|dasgupta’  emp.lst
```

```
$ grep -E ‘(sen|das)gupta’ emp.lst
```

ERE when combined with BRE forms a very powerful RE. The expression ‘agg?[ar]+wal’ contains characters from both sets.

Shell Programming
Activities of shell are not restricted to command interpretation alone.

Shell with its whole set of internal commands can be considered as a language.

A shell program executes in *interpretive* mode.

Each statement will be loaded into memory when it is executed. Shell scripts consequently run slower than those written in high-level languages.

Shell programs are considered as powerful, because the external Unix commands blend easily with the shell's internal constructs.

Shell Programming

Ordinary and Environment Variables

Variables play an important role in every programming language. In Unix two types of variables are used 1. Ordinary 2. Environment variables

A variable in a shell script is a means of **referencing** a **numeric** or **character value**.

Unlike formal programming languages, a shell script doesn’t require users to **declare a type** for variables.

Environment variables

These are the variables which are created and maintained by **Operating System itself.**

Generally these variables are defined in **CAPITAL LETTERS**.

Environment variables list can be seen using the command “**set**“.

Shell Programming
Environment variables

System defined variables	**Meaning**
BASH (/bin/bash)	Shell Name
BASH_VERSION (4.1.2(1))	Bash Version
COLUMNS (80)	No. of columns in screen
HOME (/home/dev)	Home Directory of User
LINES (25)	No. of lines in screen
LOGNAME (dev)	dev logging name
OSTYPE (linux-gnu)	OS type
PATH (/usr/bin:/sbin)	Path Settings
PWD, SHELL, USERNAME, PS1	

Shell Programming

Introduction

When a group of commands have to be executed regularly, they will be stored in a file, and the file is executed as a **shell script** or **shell program**.

File which contains shell scripts can have .sh extension (not mandatory).

Shell scripts are executed in a separate child process, created by shell. By default, the child and parent shell belong to the same type, but the first interpreter line of the script can be used to convey which shell to be used.

Ex: If login shell is Bash, Korn shell can be used to execute a shell script.

```
$ cat > script.sh
        #! /bin/sh
        echo “Today’s date: `date`”
        echo “My shell: $SHELL” ^D
```

Shell Programming

Introduction

Line which starts with # is a comment line except the first one which starts with #!, which requests the shell interpreter to choose the shell in which the following commands will be executed.

By default execute option will not be provided for the files that are created in unix. Files that contain shell scripts must have x option for execution, which will be done using chmod command.

```
$ chmod u+x script.sh
$ ./script.sh                    ‘ ./ ’ is conveying to the executor that the file is in pwd.
```

Explicitly execution of script in a different shell

```
$ ksh    script.sh
```

First line in the script will be neglected when executed in this manner.

Shell Programming : Environment variables

Values of environment variables can be displayed using echo command

```
$ echo $OSTYPE
```

Ordinary variables

These variables are defined by **users**.

Variables allows to **store data temporarily** and use it throughout the script.

User variable names can be any text string of up to **20 letters**, **digits**, or **an underscore character**.

User variables are case sensitive, so the variable name **Var1** is different from the variable name **var1**.

Values are assigned to user variables using an **equal sign.** No spaces can appear between the variable, the equal sign, and the value.

```
var1=10      var2=-57        var3=testing        var4=“still more testing”
```

Shell Programming

Ordinary variables

The shell script **automatically determines data type** of the variable.

Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes its execution.

```
        $ cat > test3.sh
                        #! /bin/bash
                        # testing variables
                        days=10
                        guest=”Katie”
                        echo “$guest checked in $days days ago”
                        days=5
                        guest=”Jessica”
                        echo “$guest checked in $days days ago”^D^D

$ chmod u+x  test3.sh           $ ./test3.sh
```

Shell Programming

Ordinary variables

Each time the variable is **referenced**, it produces the value currently assigned to it.

When referencing a variable's value, **dollar sign** has to be used. When a value is assigned to a variable dollar sign must not be used.

The .profile file

In Unix, .profile files contains definitions for a shell environment, such as environment variables, scripts to execute, and other instructions; used for storing pre-defined settings that are loaded when a shell program starts.

When opening a terminal(bash shell) in Unix, the program automatically searches for a ".profile" / ".bash_profile" file and executes it line by line as a shell script.

To manually run a profile file, use the command **source ~/.profile.** (source ~/.bash_profile)

Shell Programming

The .profile file

PROFILE files are hidden files that do not have a filename prefix. They are always named .profile and are located in the user's home directory.

To manually run a profile file, use the command **source ~/.profile.**

The .profile file is present in home ($HOME) directory and can be customized for individual working environment.

read: Making Scripts Interactive

read statement is used to accept input from the user, making scripts interactive.

Inputs read from the standard input device is placed in a variable.
Ex: read name
Script will pause at that point to accept input from the keyboard. Entered value will be stored in name.

Shell Programming

read: Making Scripts Interactive

Ex: read name

Since accepting a value is a form of assignment, $ will not be used before name.

```
$ gedit   cfile.sh
          #! /bin/sh
          echo "Enter the word to be searched: \n"
          read pname
          echo "Enter the file to be used: \c"
          read fn
          echo "Searching for $pname from file $fn"
          grep "$pname"    $fn
                           # pattern has to passed in " " for grep, if more than 1 word is considered in
          pattern.
          echo "Selected records are shown above"
```

Above script will search for the content entered in pname in the file name entered in fn.

Shell Programming

read: Making Scripts Interactive

Ex: read   pname   filename

Read statement can be used to read multiple values from the same statement.

If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned.

When the number of arguments exceeds the number of variables, the remaining words are assigned to the last variable.

```
#! /bin/sh
echo "Enter the word to be searched: \n"
read pname fname
echo $pname  $fname
```

Input: first	pname=first fname=” ”	**Output:** first
Input: first second word	pname=first fname=second word	**Output:** first second word

Shell Programming

Read only variables

Shell provides a way to mark variables as read-only by using the **read only** command. After a variable is marked read only, its value cannot be changed.

```
#! /bin/sh
readonly  cons="abc"
echo $cons
cons="iop"      # this generates while executing the script.
```

Command Line Arguments

Shell scripts accept argument from the command line. Passing information or input to script from command line is considered as noninteractive.

When arguments are specified with a shell script, they are assigned to certain special variables.

Shell Programming : **Command Line Arguments**

First argument passed from command line while executing shell script, will be read into the variable $1, second argument into $2. These are termed as **positional parameters**.

Shell Parameter	**Significance**
$1, $2, ….	Positional parameters representing command line arguments
$#	Number of arguments specified in the command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
$?	Exit status of the last command
$$	PID of the current shell
$!	PID of the last background process
"$@"	Each quoted string treated as a separate argument (recommended over $*)

Shell Programming
Command Line Arguments

```
        #! /bin/sh
        echo "Program:  $0
        The number of arguments specified is $#
        The arguments are $*"
        grep  "$1"    $2
        echo "\nJob over"

$ chmod u+x test.sh
$ ./test.sh   malloc     a.c
```

Shell Programming

exit and Exit status of a command

Shell script uses exit command to terminate the execution.

exit 0	used when no error occurred
exit 1	used when error occured

Ex: If a grep command could not locate a pattern, error message will be displayed by the grep command. grep code will invoke exit(1) command, where in value 1 is communicated to the calling program, usually the shell.

It's only through exit, that every command returns an **exit status** to the caller.

A command is said to return a **true** exit status if it executes successfully, and **false** if it fails.

```
$ cat foo
Cat:  can't open foo
```

Returns a nonzero exit status because it could not open the file. The shell offers a variable ($?) and a **test** command that evaluates a command's exit status.

Shell Programming
exit and Exit status of a command

```
#! /bin/sh
echo “first statement in script”
echo “second statement
```

```
$ ./scripts.sh
Suitable Error message will be displayed
$ echo $?
2
```

this is the exit status of scripts.sh file while executing, if the script file is successful in execution output from echo $? command will be 0

The parameter $?
Parameter $? Stores the exit status of the last command. It has the value 0 if the command succeeds and a nonzero value if it fails. This parameter is set by **exit**’s argument.

Shell Programming
exit and Exit status of a command
The parameter $?

```
$ cat > file
director
^D
$grep  director  file > /dev/null;  echo $?
0
$grep  dir  file1 > /dev/null;  echo $?
2                                                  ; Failure in opening file "file1"
```

Exit status is extremely important for a program, because the program logic, that branches into different paths are depended on the success or failure of a command.

Shell Programming

Logical operators && and || for Conditional Execution

Shell provides 2 operators that allow conditional execution of the script, && and ||.

Syntax: cmd1 && cmd2   cmd1 || cmd2

In && operator cmd2 will be executed only when cmd1 succeeds.
In || operator cmd2 is executed only when the first fails.

```
$ grep   director   file  &&  echo “Pattern found”
director
Pattern found
```

```
$ grep  direcr   file1    &&   echo ”Pattern found”    ;output will be empty
```

```
$ grep direcortry   file  ||    echo  ”Pattern not found”
Pattern not found
```

Shell Programming
Logical operators && and || for Conditional Execution

```
#! /bin/sh
echo "Enter keyword to find"
read kw
echo "Enter filename to find in"
read fn
grep "$kw"  $fn  || exit 2
echo "Pattern found"                    ; Input:  malloc      a.c      Output: Pattern found

$ vi  cfile.sh
#! /bin/sh
grep  "$1"    $2  || echo "Pattern not found"
exit 2                                  ; value 2 if passed to another program, which can be used
                                          logically, but shell received the value and updates $?
                                          Environment variable.
```

Shell Programming

The if Conditional

The if statement makes two way decisions depending on the consideration of a condition.

Shell uses the following forms of if condition.

if *command is successful*	if *command is successful*	if *command is successful*
then	then	then
execute commands	*execute commands*	*execute commands*
else	fi	elif *command is successful*
execute commands		then …
fi		else …
		fi
Form 1	**Form 2**	**Form 3**

Shell Programming : **The if Conditional**

If considers the success or failure of the command that is specified in front of it.

If the command succeeds, the sequence of commands following if is executed.

If the command fails, then the commands in else part is executed.

Statements of else part is not compulsory.

Every if is closed with a corresponding fi, and error will be encountered if “fi” is not present.

```
$ vi  test.sh
#! /bin/sh
if    grep   "$1"    /etc/passwd   2>/dev/null
then
    echo "Pattern found - Job Over"
else
    echo "Pattern not found"
    cxit 1   # if not used then “echo $?” returns 0, which represents success of test.sh
fi
```

$ ./test.sh   ftp   **$ ./test.sh   ftp1**

Shell Programming : The if Conditional

```
$ ./test.sh     ftp
ftp:*.325…….
Pattern found - Job over
```

```
$ ./test.sh      abcd
Pattern not found
```

The test command and its shortcut

The test utility evaluates the expression and, if it evaluates to true, returns a zero (true) exit status; otherwise it returns 1 (false). If there is no expression, test also returns 1 (false).

test command will be used to convert the value generated by relational expression, in a manner that can be interpreted by ‘if’ command.

test command works in 3 ways:

Compare two numbers

Compares two strings or a single one for a null value.

Checks a file’s attributes

Shell Programming
The test command and its shortcut
‘test’ command will not display any output, but simply sets the parameter $?

Numeric Comparison
Numerical comparison operators used by test has an - (hyphen), followed by a two-letter string, and enclosed on **either side by whitespace**.

Operator	**Meaning**
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

Shell Programming

Numeric Comparison

Numeric comparison in the shell is confined to integer values only; decimal values are simply neglected.

```
$ x=5;  y=7;  z=7.2
$ test $x -eq $y  ;  echo $?
1                                  Not equal
$ test $x -lt $y  ;  echo $?
0                                  True
$ test $z -eq $y   ; echo $?
bash: test: 7.2: integer expression expected
2                                  Error
```

```
$ vi   test.sh
#! /bin/sh
if  test  $#  -eq  0
then
        echo "Usage: $0 <pattern> <file>"
elif  test  $#  -eq  2
then
        grep "$1"  $2  ||  echo "$1 not found in $2"
else
        echo "Two arguments are required"
fi
```

```
$ ./test.sh  printf    add.c

$ ./test.sh     prif     add.c
prif not found in add.c
```

Shell Programming : **Shorthand for test**

test command is widely used, hence there is a shorthand method of executing it. A pair of rectangular brackets enclosing the expression can be used instead of test command.

```
Ex:    test    $x  -eq  $y
        [ $x -eq $y ]
```

Whitespace is mandatory around the operator

String Comparison

test can be used to compare strings. String Tests Used by **test**

Test	**True if**
s1 = s2	String s1 = s2
s1 != s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string

Test	**True if**
stg	String stg is assigned and not null
s1 == s2	String s1 == s2 (Korn and Bash only)

Shell Programming : String Comparison

```
$  vi  test1.sh
#! /bin/sh
if [ $# -eq 0 ]
then
        echo "Ente the string to be searched: \c"        # \c to make cursor remain in the same line.
        read pname
        if [ -z "$pname" ]                              # checking whether pname is null or not
        then
                echo "No string entered"
                exit 1                                  # exit by setting $? to 1
        fi
        echo "Enter filename: \c"
        read filename
        if [ ! -n "$filename" ]                  # -n checks for not null string, ! negates the result
        then
                echo "No string entered"
                 exit 1
        fi
        ./test.sh "$pname"  "$filename"          # executing the previous script test.sh  './' is compulsory
else
        ./test.sh   $*                           # executing  test.sh, by passing complete set of positional parameters
fi
```

Shell Programming : String Comparison

```
$ ./test1.sh   “Jai sharma”  xz.c
```

Assume Jai sharma is the name of the programmer who created xz.c and this name is present in the comment line in xz.c. Output of the above command will be

Two arguments are required

Reason being Jai and sharma are embedded in $* as separate arguments.

$# thus makes a wrong argument count.

Solution is to replace the line test.sh $* in the previous program by test.sh “$@”

```
# t.sh                 # t1.sh            # t.sh                # t1.sh
#! /bin/sh             #! /bin/sh         #! /bin/sh            #! /bin/sh
./t1.sh $*             echo $#            ./t1.sh  “$@”         echo $#

Input:                 Output:            Input:                Output:
./t.sh  “a b”  x.c     3                  ./t.sh  “a b” x.c     2
```

Shell Programming : **String Comparison**

test command also permits checking of more than one condition, using -a (AND) and -o (OR) operators.

```
#! /bin/sh
echo "Enter 1st string"
read str1
echo "Enter 2nd string"
read str2
# if [ -n "$str1" -a -n "$str2" ]
if [ -n "$str1" ] && [ -n "$str2" ]          # not necessary to enclose $str1 in " "
then
        if [ "$str1" = "$str2" ]
        then
                echo "Strings are equal" ; exit 0
        else
                echo "String are not equal" ; exit 0
        fi
fi
echo "You have entered a null string"
```

Shell Programming : **File Tests**

test command can also be used to test various file attributes like its type (file, directory or symbolic link) or its permissions (read, write, execute, etc.,)

Test	**True if File**
-f *file*	*file* exists and is a regular file
-r *file*	*file* exists and is readable
-w *file*	*file* exists and is writable
-x *file*	*file* exists and is executable
-d *file*	*file* exists and is a directory
-s *file*	*file* exists and has a size greater than zero
-e *file*	*file* exists (Korn and Bash only)
-u *file*	*file* exists and has **SUID bit** set
-k *file*	*file* exists and has **sticky bit** set

Test	**True if File**
-L *file*	*file* exists and is a symbolic link (K & B only)
f1 -nt f2	f1 is newer than f2 (K & B only)
f1 -ot f2	f1 is older than f2 (K & B only)
f1 -ef f2	f1 is linked to f2 (K & B only)

Shell Programming : File Tests

```
#! /bin/sh
echo $0
if [ ! -e $1 ]
then
        echo "File does not exist"
elif [ ! -r $1 ]
then
  echo "File is not readable"
else
  echo "File is readable"
fi
if [ ! -w $1 ]
then
  echo "File is not writable"
else
  echo "File is writable"
fi
```

Shell Programming

The case Conditional

case is the second conditional statement used by shell.

case statement matches an expression for more than one alternative and uses a compact construct to permit multiway branching. case also handles string tests.

Syntax:

```
case  expression  in
        pattern1)   commands1  ;;
        pattern2)   commands1  ;;
        pattern3)   commands1  ;;
        ….
esac
```

case first matches *expression* with *pattern1*. If match succeeds, then it executes *commands1*, which may be one or more commands. If match fails, then *pattern2* is matched, and so forth.

Each command list is terminated with a pair of semicolons, and the entire construct is closed with esac (reverse of case).

Shell Programming
The case Conditional

```
#! /bin/sh
echo "           MENU\n
1. List of files\n2. Processes of user\n3. Todays Date
4. Users of system\n5. Quit to SHELL\nEnter your option: \n"

read choice

case "$choice" in
        1) ls -l     ;;
        2) ps -f     ;;
        3) date      ;;
        4) who       ;;
        5) exit      ;;
        *) echo "Invalid option"
esac
```

Shell Programming : The case Conditional

case cannot handle relational and file tests, but it matches strings with compact code.

```
#! /bin/sh
read str
case "$str" in
        Mon)   echo "Monday"           ;;
         Tue)   echo "Tuesday"           ;;
        Wed)   eco   "Wednesday"      ;;
             *)    echo "Invalid option"
esac
```

Matching Multiple Patterns

case can specify the same action for more than one pattern, case uses the | to delimit multiple patterns.

```
Ex:  #! /bin/sh
        echo "Continue (y|n): \n"   ;  read  ans
        case "$ans" in
                y|Y) echo "Choice is yes" ;;
                n|N) echo "choice is no" ; exit ;;
        esac
```

Shell Programming : The case Conditional

Matching Multiple Patterns

case has a string matching feature which uses of wild-cards.

case uses the filename matching metacharacters *, ? and the character class, to match strings and not files in the current directory.

Ex:
```
#! /bin/sh
read ans
case "$ans" in
        [yY][eE]*)   echo "In ye*/YE* part" ; echo $ans  ;;
         [nN][oO])   echo $ans  ; exit ;;
               *)   echo "Invlaid Output"
esac
```

* bheaves in two different manners.

In the first step, it behaves like a normal wild-card manner,

In the second step it acts as a last option for all other non-matched option.

Shell Programming
while Looping

Loops helps to execute a set of instructions repeatedly. Shell features 3 types of loops - **while, until** and **for**.

While loop executes a set of instructions until the control command returns a true exit status.
Syntax:

```
    while  condition-is-true
    do
        commands
    done
```

Commands enclosed by **do** and **done** are executed repeatedly as long as *condition* remains true.
Any Unix command or **test** can be used as the *condition*.

Shell Programming
while Looping

```
#! /bin/sh
# accepting student name, usn and storing it in a file until ‘n/no/NO..’ is pressed
ans=y
while [ $ans = "y" ]
do
        echo "Enter student name and usn"
        read name usn
        echo "$name | $usn" >> newlist     # >> redirection operator for appending
        echo "Continue (yes/no)"
        read ch  echo $ch
        case "$ch" in
                [yY][eE][sS])    ;;                    # matches YEs, yes, Yes, YEs, yES, etc
                     [nN][oO]) exit ;;                 # matches NO, no, nO and No
                               *) echo "Invalid option"
        esac
done
```

Shell Programming

while Looping

Symbol >> in the previous program, opens “newlist” file each time when echo command is executed.

Shell avoids such multiple file openings and closures by providing a redirection facility at the done (of while-do-done) keyword itself: **done >> newlist**

This form of redirection speeds up execution time as the file newlist is opened and closed only once and this operator redirects the standard output of all commands inside the loop to the file.

Some statements can be redirected to /dev/tty, for user interaction.

Redirection is also available at the fi and esac keywords, and includes input redirection.

Ex:	done < param.lst	Statements in loop take input from param.lst
	fi >> foo	Affects statements between if and fi
	esac >> foo	Affects statements between case and esac.

Shell Programming
Using while to Wait for a File

```
#! /bin/sh
while [ ! -r newlist ]
do
    sleep 60
done
echo "File created"
```

Above script waits until newlist file is created, command to execute this script is **'./try1.sh &'**, **&** will make the script to execute in background.

sleep command makes the script pause for the duration (in seconds) as specified in its argument.

Loop executes repeatedly as long as the file newlist is not available and cannot be read(! -r means not readable). If the file is available and is readable then loop terminates and shell stops executing.

Shell Programming
Setting Up an infinite Loop

```
#! /bin/sh
while true
do
        echo “Take a break”
        sleep 30
done &          # & after done runs loop in background
```

while true returns a true exit status. Another command named **false** returns a false value.

ps is the command to see the processes that are currently being executed.

kill <PID> is the command to stop the execution of a specific process, using its PID.

Shell Programming - for: Looping with a list

for looping construct in shell script will **not use condition to iterate**, but uses a list instead.

Syntax: **for** *variable* **in** *list*

```
        do
            commands                    # loop body
        done
```

For loop uses, do and done keyword as in while construct, but the additional parameters are *variable* and *list*.

Each whitespace-separated word in list is assigned to variable in turn, commands are executed until *list* is exhausted.

Ex:
```
#! /bin/sh
for  file  in  k1.c  k22.c  k2.c  k5.c      # list of filenames, separated by whitespaces.
do                                         # Each item in the list is assigned to the variable file
    cat $file                              # 'file' first gets value k1.c, then k22.c and so on in each
    echo "Press any key to continue..."  # iteration.
    read cont
done
```

Shell Programming
for : Possible Sources of the List

List can consist of practically any of the expressions that the shell understands and processes.

List from variables: A series of variables of command line can be used in list.

Ex:
```
#! /bin/sh
for var in $PATH $HOME $MAIL
do
        echo $var
        echo "Press any key to continue..."
        read cont
done
```

List from command substitution: Command substitution can be used to create a list. Following example picks up its list from the file clist.

Ex:
```
#! /bin/sh
  for var in `cat clist` ; do
        $var
        echo "Press any key to continue..."
  read cont  ; done
```

```
$ cat > clist
ls
date ^d
```

Shell Programming

for : Possible Sources of the List

List from wild-cards: When the list consists of wild-cards, shell interprets them as *filenames*.

Ex:
```
#! /bin/sh
for  fname  in  *.html
do
        echo $fname
done # lists all html files
```

```
$ cat > clist
*.html
*.c
^d
```

```
#! /bin/sh
for fname in `cat clist`
do
        echo $fname
done # lists all html files
```

List from positional parameters: Positional parameters that are passed from command line can also be accessed in for.

Ex:
```
#! /bin/sh
    for var in "$@"        # each double quotes argument is considered as one argument
    do
       echo $var
       echo "Press any key to continue..."
    read cont  ; done
```

Shell Programming : **set**

set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg ...]

set command is used to set or unset values of shell options and positional parameters.
set assigns the arguments passed to it, to the positional parameters $1, $2 and so on.
Value of shell attributes can be changed using set command.
This feature helps to access individual fields from the output of a program, such as date or any other command.

```
$ set 98 76 54
```

This, assigns the value 98 to the positional parameter $1, 76 to $2, 54 to $3. It also sets other parameters $#(argument count) and $*(all arguments).

```
$ echo $1  $2  $3

$ set `date`              # will extract individual fields from date output

$ echo $1  $2  $3  $4
Tue  Nov  9  09:47:22
```

Shell Programming : set

$ set `ls -l x.c`

Output of ‘ls -l x.c’ starts with a ‘-’ (--rw-r--r-- can be the first output of ls -l x.c), since the permission string begins with a -, set interprets it as an option and flags an error.

Another, problem with set will be when a command generates null output.

$ set `ls -l ui` considering file ui is not present in pwd, this command generates some random output.

$ set `grep PPP /etc/pwd` considering PPP is not available in pwd file, output of this command will be other contents of pwd.

To avoid these two situations -- option will be used with set command.

```
$ set -- `ls -l x.c`  ;  echo $#
9             9 represents, the positional parameters that are assigned by set command
$ set -- `grep PPP /etc/pwd` ; echo $#
0             because PPP is not found in pwd file.
```

Shell Programming

shift

shift transfers the contents of a positional parameters to its immediate lower numbered one. Shift of values will be done as many times as the statement is called.

When called once, $2 value will be transferred to $1 and $1 value will be overwritten.
 Called second time, $3 value will be transferred to $2.

```
$ set 98  76  54  32
$ echo $*
98  76  54  32

$ shift
$ echo $1  $2   $3   $4
76  54  32
```

```
$ set 98  76  54  32
$ echo $*
98  76  54  32

$ shift 2
$ echo $1  $2   $3   $4
54  32
```

Shell Programming

Manipulating the positional parameters

```
#!/bin/sh
case   $#   in
    0|1)  echo "Usage: $0 file pattern(s)"
          exit 2 ;;
      *)  fn=$1
          shift
          for pattern in "$@"
          do
             grep -n "$pattern" $fn || echo "Pattern $pattern not found"
          done ;;
esac
```

```
$  ./stu.sh   data.txt     stu1   stu2   40  stu3
```

First CLA must be file name in which pattern has to be found.
(-n option to print the line number if the pattern is found.)

Shell Programming

The HERE document (<<)

A *here document* is a special-purpose code block. It uses a form of I/O redirection to feed command list to an interactive program.

Syntax:

```
COMMAND << InputEndsHERE
...
...
...
InputEndsHERE
```

A *limit-string (i.e InputEndsHERE in this case)* represent data list. The special symbol << precedes the limit string. This has the effect of redirecting the output of a command block into the stdin of the program or command.

It is similar to interactive-program < command-file, where command-file contains

```
                command #1
                command #2
                ……                    which represents data
```

Shell Programming
The HERE document (<<)

limit-string has to be chosen sufficiently unusual that it will not occur anywhere in the command list, which might lead to confusion.

```
#!/bin/sh
echo "Enter file name"
read fname
echo "Enter pattern"
read pat
grep "$pat" $fname || echo "Pattern $pat in $fname not found"
```

```
$ ./try.sh  << end
> data.txt
> rns
> end
```

Here, “end” is the *limit-string*.

Shell Programming

trap: Interrupting a program

There might be situations when users of a script must not perform untimely exit using keyboard abort sequences, because input might be provided or cleanup has to be done.

trap statement catches these sequences and can be programmed to execute a list of commands upon catching those signals. Syntax of **trap** statement is.

trap [COMMANDS] [SIGNALS]

This instructs the **trap** command to catch the listed *SIGNALS*, which may be signal names with or without the *SIG* prefix, or signal numbers.

If a signal is *0* or *EXIT*, the **COMMANDS** are executed when the shell exits.

If one of the signals is *DEBUG*, the list of **COMMANDS** is executed after every simple command.

Shell Programming
trap: Interrupting a program

A signal may also be specified as *ERR*; in that case **COMMANDS** are executed each time a simple command exits with a non-zero status.

Note that these commands will not be executed when the non-zero exit status comes from part of an **if** statement, or from a **while** or **until** loop. Neither will they be executed if a logical *AND* (&&) or *OR* (||) result in a non-zero exit code, or when a command's return status is inverted using the *!* operator.

```
#!/bin/sh                                           # trap  -l  will list all signals
trap ' echo has pressed ctrl+c/kill ; exit '  2 15  # 2 - SIGINT    15 - SIGTERM
echo "pid is $$"
read one
```

Press ctrl+c which generates SIGINT (2) and trap command echo is executed first and then exit will be executed finally. Type “kill <pid>” to generate SIGTERM signal.

Shell Programming

trap: Interrupting a program

Signals generated during the execution of script can be ignored using the following trap command.

```
trap ‘ ‘ 1 2 15
```

Shell script to print nos 1 to 50 and when ^c is pressed value printed will increment by 5

```
#! /bin/sh
n=1
trap 'n=$((n+5))' 2
while test $n -lt 50
do
   echo $n
   sleep 2
   n=$((n+1))
done
```

Shell Programming : To find the dept name by accepting dept code

```
#! /bin/sh
IFS="|"  # Internal Field Separator can be <space>, *, $ ...
while  echo "Enter department code : \c"
do
        read dcode
        set -- `grep $dcode << limit
```

# ^ represents to check $dcode pattern at the beginning of the line

```
01|accounts|1
02|marketing|2
03|personnel|3
04|product|4
limit`
        case $# in
                3) echo "Department name: $2\nEmp-id of head of dept: $3\n"
                shift 3 ;;
                *) echo "Invalid code"
        esac
done
```

Shell Programming

Program to print numbers sequentially and to set a trap for ^c upon receiving ^c signal, value will be incremented by 5

```
#! /bin/sh
n=1
val=2000
trap 'n=$((n+5))' 2
while test $n -lt 50
do
   echo $n
   sleep 2
   n=$((n+1))
done
```

Shell Programming

Program to print numbers sequentially and to set a trap for ^c upon receiving ^c signal, value will be incremented by 5 - using functions

```
#! /bin/sh
trap 'increment' 2
increment( )
{
  echo "caught signal"
  x=`expr $x + 6`
  if [ $x -gt "100" ]
  then
     echo "okay i will quit"
  exit 1
  fi
}
```

Shell Programming

Program to print numbers sequentially and to set a trap for ^c upon receiving ^c signal, value will be incremented by 5 - using functions

```
x=0
while :
do
  echo $x
  x=`expr $x + 1`
  sleep 1
done
```

When the program goes beyond 100, and when ^c is pressed increment function is called and if condition terminates the process.

Shell Programming

```
$ cat > desig.lst
01 director
02 principal
03 hod
04 staff^D
```

SIGHUP ("signal hang up") is a signal sent to a process when its controlling terminal is closed.

```
$ kill -1 <pid>
```

Shell Programming

```
#!/bin/sh
trap 'echo Not to be interrupted' INT
trap 'echo Signal received ; exit ' HUP

file=desig.lst
while echo "Designation code: \c" > /dev/tty
do
    read desig
    case "$desig" in
          [0-9][0-9]) if grep "^$desig" $file > /dev/null
                             then
                                 echo "Code exists"
                                 continue
                              fi;;
                        *) echo "Invalid code"
                           continue;;
    esac
```

Shell Programming

```
        while echo "Description: \c" > /dev/tty
        do
            read desc
            case "$desc" in
                 *[!a-zA-Z]*) echo "Can contain only alphabets and spaces" > /dev/tty
                                   continue ;;
                             " ") echo "Description not entered" > dev/tty
                                  continue ;;
                              *) echo "$desig $desc"
                                  break
            esac
        done >> $file
        echo "\nWish to continue? (y/n): \c" ;  read answer
        case "$answer" in
            [yY]*) continue ;;
                *) break ;;
        esac
     done ;      echo "Normal exit"
```