UNIX PROGRAMMING

MODULE-2

THE BASIC FILE ATTRIBUTES

ls –l : listing file attributes

The **ls** command lists files and directories. If the *pathname* is a file, **ls** provides information according to the specified options. If *pathname* is a directory, **ls** lists the files and subdirectories contained in the specified directory. Information about the directory itself can be obtained using the **-d** option.

If no *pathname* is provided, **ls** displays the files and directories in the current directory.

```
$ ls -l
total 32
-rw-rw-r-- 1 anjan anjan   9 Feb 14 10:02 d
-rw-rw-r-- 1 anjan anjan   6 Feb 14 09:47 dep
-rw-rw-r-- 1 anjan anjan  72 Feb 14 09:54 dept.lst
-rw-rw-r-- 1 anjan anjan  26 Feb 14 10:07 f1
-rw-rw-r-- 1 anjan anjan  26 Feb 16 03:41 f2
-rw-rw-r-- 1 anjan anjan  26 Feb 16 03:46 f3
-rw-rw-r-- 1 anjan anjan  26 Feb 14 10:03 file1
-rw-rw-r-- 1 anjan anjan   0 Feb 14 10:04 file2
-rwxrwxr-x 1 anjan anjan 548 Feb 14 10:16 pswd_chk.html
```

- The first column gives the type of the file (*e.g.*, directory or ordinary file) and the file permissions.
- The second column is the number of links to the file *i.e.,* (more or less) the number of names there are for the file. Generally an ordinary file will only have one link, but a directory will have more, because you can refer to it as ``dirname", ``dirname/." where the

dot means ``current directory'', and if it has a subdirectory named ``subdir'', ``dirname/subdir/..'' (the ``..'' means ``parent directory'').

- The third and fourth columns are the user who owns the file and the Unix group of users to which the file belongs. You almost certainly don't need to worry about Unix groups, as you probably only belong to the default group ``users''.
- The fifth column is the size of the file in bytes.
- The next three columns are the time at which the file was last changed (for a directory, ths is the time at which a file in that directory was last created or deleted).
- The last column is the name of the file.

Prefix	Description
-	Regular file, such as an ASCII text file, binary executable, or hard link.
b	Block special file. Block input/output device file such as a physical hard drive.
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories.
l	Symbolic link file. Links on any regular file.
p	Named pipe. A mechanism for interprocess communications
s	Socket used for interprocess communication.

The –d option:

display information about named directory, rather than directory contents

ls –ld dir1 dir2

drwxr-xr-x 1 anjan anjan 9 Feb 14 10:02 dir1

drwxr-xr-x1 anjan anjan 6 Feb 14 09:47 dir2

FILE OWNERSHIP

When you create a file, you will become the owner of the file. Several users may belong to same group. When the system admin creates a user account, he has to assign these parameters to the user:

- The user id (UID)
- The group id(GID)

The file /etc/passwd maintains UID

The file /etc/group maintains GID

To know your UID and GID then

$id

Uid=4532 gid=7532

FILE PERMISSIONS

Unix - File Permission

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

- **Owner permissions:** The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions:** The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions:** The permissions for others indicate what action all other users can perform on the file.

$ ls -l

total 32

-rw-rw-r-- 1 anjan anjan 9 Feb 14 10:02 d

-rw-rw-r-- 1 anjan anjan 6 Feb 14 09:47 dep

-rw-rw-r-- 1 anjan anjan 72 Feb 14 09:54 dept.lst

-rw-rw-r-- 1 anjan anjan 26 Feb 14 10:07 f1

-rw-rw-r-- 1 anjan anjan 26 Feb 16 03:41 f2

-rw-rw-r-- 1 anjan anjan 26 Feb 16 03:46 f3

-rw-rw-r-- 1 anjan anjan 26 Feb 14 10:03 file1

-rw-rw-r-- 1 anjan anjan 0 Feb 14 10:04 file2

-rwxrwxr-x 1 anjan anjan 548 Feb 14 10:16 pswd_chk.html

Here first column represents different access mode ie. permission associated with a file or directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x):

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that onwer has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--** represents that group has read (r) and execute (x) permission but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

File Access Modes: The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which are described below:

Read: Grants the capability to read ie. view the contents of the file.

Write: Grants the capability to modify, or remove the content of the file.

Execute: User with execute permissions can run a file as a program.

Directory Access Modes: Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

Read: Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

Write: Access means that the user can add or delete files to the contents of the directory.

Execute: Executing a directory doesn't really make a lot of sense so think of this as traverse permission.

Chmod: changing the file permissions

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: relative mode and absolute mode.

i. Relative permissions

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here is an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

$ls -l testfile

-rwxrwxr-- 1 anjan users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes: [anjan]$chmod o+wx testfile

$ls -l testfile

-rwxrwxrwx 1 anjan users 1024 Nov 2 00:10 testfile

$chmod u-x testfile

$ls -l testfile

-rw-rwxrwx 1 anjan users 1024 Nov 2 00:10 testfile

$chmod g=r-x testfile

$ls -l testfile

-rw-r-xrwx 1 anjan users 1024 Nov 2 00:10 testfile

Here is how you could combine these commands on a single line:

$chmod o+wx,u-x,g=r-x testfile

$ls -l testfile

-rw-r-xrwx 1 anjan users 1024 Nov 2 00:10 testfile

ii. ***Absolute Permissions:***

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here is an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

$ls -l testfile

-rwxrwxr-- 1 anjan users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:

$ chmod 755 testfile

$ls -l testfile

-rwxr-xr-x 1 anjan users 1024 Nov 2 00:10 testfile

$chmod 743 testfile

$ls -l testfile

-rwxr---wx 1 anjan users 1024 Nov 2 00:10 testfile

$chmod 043 testfile

$ls -l testfile

----r---wx 1 anjan users 1024 Nov 2 00:10 testfile

Changing Owners and Groups:

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files:

1. **chown:** The chown command stands for "change owner" and is used to change the owner of a file.

2. **chgrp:** The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership:

The chown command changes the ownership of a file. The basic syntax is as follows:

$ chown user filelist

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system

Following example: [anjan]

$ chown anjan testfile

$

changes the owner of the given file to the user **anjan**.

NOTE: The super user, root, has the unrestricted capability to change the ownership of a any file but normal users can change only the owner of files they own.

Changing Group Ownership:

The chrgp command changes the group ownership of a file. The basic syntax is as follows:

$ chgrp group filelist

The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

Following example:

$ chgrp special testfile

$

Changes the group of the given file to **special** group

SECURITY IMPLICATIONS

Default permissions for the newly created file is 644, for example

-rw-r—r-- 1 anjan users 1024 Nov 2 00:10 testfile

Here only user can edit no one else, if we remove all then

Chmod u –rw,go –r testfile

Or

Chmod 000 testfile

Then

---------- 1 anjan users 1024 Nov 2 00:10 testfile

We can't do anything on this except delete.

In other hand

Chmod u +x,go+wx testfile

Or

Chmod 777 testfile

Then

-rwxrwxrwx 1 anjan users 1024 Nov 2 00:10 testfile

This is also dangerous because anyone can do anything to the testfile.

USING CHMOD RECURSIVELY

It's make chmod descend a directory hierarchy and apply the expression to every file and subdirectory. This is done by using –R option.

Chmod –R 755 . works on hidden files also

Chmod –R a+x * leaves out hidden files

DIRECTORY PERMISSIONS

Directories also have their own permissions, default permissions for newly created directories is 755 .

drwxr-xr-x 1 anjan users 1024 Nov 2 00:10 vtu

SHELL INTERPRETER

When you input a command, the shell first scans the command line for metacharacter. These are called special characters.

For example

cat > vtu

rm *

ls | more

Here metacharacters >,*,| are special meaning,

The following activities are performed by the shell in its interpretive cycle.

i. The shell issues the prompt and waits command to enter.
ii. The shell scans the command line for metacharacters.
iii. It passes command line to the kernel for execution
iv. Shell waits command to complete and normally can't do any work while the command is running
v. Shell starts next cycle

SHELL OFFERING:

In UNIX there are two major types of shells:

i. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
ii. The C shell. If you are using a C-type shell, the default prompt is the % character.

PATTERN MATCHING

We need patterns to describe text that is inexact, such as 'all words starting with s', 'all words starting with t and ending with m', or 'all 9 digit numbers'. We might be able to list all the possible variations, but it would be impractical and annoying.

We use a pattern matching language to describe a pattern. A pattern matching language uses symbols to describe both normal characters (that are matched exactly) and meta characters (that

describe special operations, such as alternatives and repeating sections). Patterns are almost always described using what is called a *regular expression. Below table illustrates the wild-card set and their usage*

Wild-card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character either an I,j or k
[x-z]	A single character is with in the ASCII range of x and z
[!ijk]	A single character that is not an I,j or k
[!x-z]	A single character is not with in the ASCII range of x and z
{pat1,pat2,…}	Pat1,pat2,etc…

The * and ?

*:- it matches Any number of characters. For e3xample

$ls chap*

List all files starting with chap

$echo*

Displays all file names in the current directory match a solitary *.

?:-it matches a single character

$ls chap?

Chap1 chap2 chapa…etc

$ls chap??

Matches two characters after chap.

MATCHING DOT

The * doesn't match all files beginning with dot(.).

$ls .???*

.bash_prof .exrc .profile

THE CHARACTER CLASS

It comprises a set of characters enclosed by the rectangular brackets, [and], but match a single character in the class. The pattern [abcd] is a character class, and it matches a single character an a, b, c or d. this can be combined with any string or wild-card.

$ ls chap0[123]

Chap01 chap02 chap03

Range can also specified by the symbol hypon(-) for example

$ls chap0[1-3]

$ls chap0[a-z]

Negating character class:

! symbol is used to negate the class.

For example

*.[!co]	matches all filenames with a single character extension but not .c or .o file

other examples

[!a-za-z]	matches all filenames that don't begin with an alphabetic character.
ls *.c	lists all files with extension with .c
mv * ../bin	moves all files to bin subdirectory of parent directory
cp ??? progs	copies to progs directory all files with three character names.
rm *.[!log]	removes all files with three character extensions except .log extension.

ESCAPING AND QUOTING:

Some file names may contain special characters for example chap*

$ls chap*

Chap chap* chap01 chap02 chap03 chap04 chap05

Suppose we want to remove chap*

$rm chap*

It would be dangerous because it remove all above files to avoid this shell provides two methods

ESCAPING:-

Providing a backslash(\) before the wild card to remove its special meaning. For instance, in the pattern *, the \ tells the shell to remove special meaning of *.

$rm chap* it doesn't remove other than chap*

Echo > chap[1-3]

Then should escape the two rectangular brackets when accessing file ie

$ls chap\[1-3\0

Chap[1-3] must escape [and]

Escaping the space:-

The shell uses delimit command arguments. To remove My document.doc which has a space embedded.

Rm My\document.doc without \ shell assumes two files ie My and document.doc

Escaping the newline character: -

It makes the end of the command line that uses several arguments can be long enough to overflow to the next line. Ie use \ before ENTER

$cp mydocumentprogram.c/usr/bin/local/home/var/www/html/lab/visweswarayainstitute/ \[ENTER] >/student

Here cp commands escapes the special meaning of ENTER and file will copied to student Directory

QUOTING:-

This is another way to turn off the meaning of special character. when command argument is enclosed in quotes, here meaning of all enclosed special characters are turned off. We can use both single and double quotes.

$echo '\' displays \

$rm chap* removes file chap*

$rm "My document.doc" removes file My document.doc

Quoting is useful while so many special characters are in command line. For example

$echo 'the characters | , > , < and $ are special'

the characters | , > , < and $ are special

Here we can also use escaping but we need to use four times \. Double quotes are more permissive they don't protect ~(backquote).

REDIRCTION:

These are the special files are actually stream of characters which many commands see as input and output. A stream is simply sequence of bytes. When user logs in the shell makes available three files available these three represents three streams. ie

i. Standard Input:- The file representing input, which is connected to the keyboard.
ii. Standard output:- The file representing output, which is connected to the display.
iii. Standard Error:- The file representing error message that emulates from command or shell, which is connected to the Display.

Every command that uses streams will always find these files open and available. The files are closed when the command completes execution.

Standard Input:-

Cat and wc commands are used to read disk files. These have an additional method of taking input. When they are used without arguments, they read the file representing standard input. This file is special and represents three input sources.

i. The keyboard (default)
ii. The file using redirection with the < symbol(a metacharacter)
iii. Another program using pipeline.

If you use wc without argument and have no special symbol like < or | in command line.wc obtains input from default source (keyboard)

$wc

Hi

This is VTU

[ctrl-d]

2 4 14

$_

It is very similar to cat. Here enters two lines and end of input[ctrl-d]. wc which uses the stream from standard input. Immediately counts 2 lines, 4 words and 14 characters.

The shell's can reassign the standard input file to a disk file. This means it can redirect the standard input to originate from a file on disk. This redirection requires < symbol.

$wc < sample.txt

2 4 14

Here sample.txt contains 2 lines, 4 words and 14 characters respectively. Above command line does the following.

i. The < the shell opens the disk file sample.txt
ii. It unplugs the standard input file from its default source and assign it to sample.txt
iii. Wc reads from standard input

TAKING INPUT BOTH FROM FILE AND STANDARD INPUT

Here symbol – is used to indicate the sequence of taking input.ie

Cat - file	First from standard input and then from file
Cat file -- bar	First from file, then standard input and then from bar

STANDARD OUTPUT

All commands displaying output on the terminal write to the standard output as a stream of characters, and not directly write to terminal. There are three possible destinations

i. The terminal, the default destination
ii. A file using the redirection symbol > an >>
iii. As input to another program using a pipeline

Example

$wc sample.txt > newfile

$cat newfile

2 4 14

Sends word count of sample.txt to newfile. Assume that new file exist then the shell overwrite it.

To avoid to use >> symbol to append to a file

$wc sample.txt>>newfile

If you use wc sample.txt >newfile then the redirection works as follows.

i. On seeing the >, the shell opens a disk file, ie newfile for writing
ii. It unplugs standard output file from its default destination and assign it to newfile
iii. Wc opens sample.txt for reading
iv. Wc writes the standard output newfile.

STANDARD ERROR

All files are represented by a number called file descriptor. A file is opened by referring pathname, but subsequent read write operations identify the file by file descriptor. The kerner maintains a table of file descriptor for every process running on the system. The first three slots are

0- Standard input

1- Standard output

2- Standard error

These descriptors are implicitly prefixed to the redirection symbols. While opening a file (< and >) both the symbols are identical. File will be allotted the descriptor 3.

When you try to open a nonexistent file, certain diagnosis message occur thorough an default destination(standard error).

For example

$cat main

Cat: cannot open main

Here we trying to open nonexistent file called main.cat fails to open the file and writes to the standard error.

UNIX commands are grouped into four categories is as follows:

1. Directory oriented commands like mkdir, rmdir,cd etc.., and basic file handling commands like cp,mv,rm et…., use neither standard input or output.
2. Commands like ls, mv who etc.., don't read standard input but writes to standard output.
3. Command like lp that read standard input but don't write standard output.
4. Commands like cat,wc,gzip…etc , that uses both standard input and output.

The fourth category is called Filters called as text manipulator. Most filters can also read directly from files whose names are provided as arguments.

For example we redirecting bc's standard input to file called calc.txt

$cat calc.txt

2*3

2^3

2^3+2^3

Now we use bc get input from file and write output to other file is as shown below

$bc < calc.txt > result.txt uses both stdin and stdout

$cat result.txt

6

8

16

SPECIAL FILES

The /dev/null and /dev/tty are called an special files. Suppose if you want check whether the program runs successfully without seeing output on the screen means save elsewhere. We have a special file that simply accepts any stream without growing in size ie /dev/null.

$cmp file1 file2 >/dev/null

$cat /dev/null size is always Zero

$_

Second file is /dev/tty indicates terminal. This is not the file that represents standard output or error. Commands usually don't writes to this file. But you want redirect something to this then Issue command like

Who>/dev/tty

The list of current users is sent to the terminal .

PIPES:- Connecting Commnads

A Unix pipe provides a one-way flow of data. The symbol | is the Unix pipe symbol that is used on the command line. What it means is that the standard output of the command to the left of the pipe gets sent as standard input of the command to the right of the pipe. Note that this functions a lot like the > symbol used to redirect the standard output of a command to a *file*. However, the pipe is different because it is used to pass the output of a command to *another command*, not a file.

For example who command produces a list of users, let use redirection to save output

$who >user.txt

$cat user.txt

Root	console	mar	5	05:05	(:0)
vtu	pts/10	mar	5	05:55	(pc123.vtu.com)
vtu	pts/15	mar	5	07:55	(pc126.vtu.com)
vtu	pts/17	mar	5	08:55	(pc129.vtu.com)

we know that wc make complete count of this , here we use –l option to print number of lines only.

$wc –l <user.txt

4

Here we are running two commands separately, it has disadvantages ie

- For long running commands, process can be slow, second command can't act until first completes
- Need intermediate file.

To over come this use pipeline concept and above can be rewritten as

$who |wc –l

4

We can also rtedirect the output for example

$ls | wc –l > file stored in file

SHELL VARIABLES

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

Defining Variables:

Variables are defined as follows::

Variable_name=variable_value,

but its evaluation requires the $ as prefixto the variable name.

For example

$count=5	No $ required for assignment
$echo $count	but needed for evaluation

A variable can also be assigned the value of another variable.

$total=$count	Assigning a variable to another variable.
$echo $total	
5	

Unsetting Variables:

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command:

Unset variable_name

Unset x	x is now undefined
Readonly x	x can’t be reassigned

USAGE OF SHELL VARIABLES

Setting pathnames:if pathname is used several times in a script, you should assign it to a variable.

$progs='/home/kumar/c_progs'

$cd $progs : pwd

/home/kumar/c_progs

Using Command Substitution:

We can also use the feature of command substitution to set variables

$mydir=~pwd~ ; echo $mydir

/home/kumar/c_progs

We can also store the size of the file in a variable too;

Size=~wc –c < file.txt~

Concatenating variables and strings

To concatenate two variables you can either place them side by side

Base=file ; ext=.c	two assignment in one line
File1=$base$ext	This is file.c

Or use curly bases to delimit them

File1=${base}$ext

We can also run executable command

cc –o $base $file1	creates executable file from file.c

grep

cat emp.lst

2233|a.k.shukla	|g.m.	|sales	|12/12/53	|6000
9876|jai Sharma	|director	|production	|12/03/50	|7000
5678|sumit chakrabarty	|d.g.m.	|marketing	|19/04/43	|6000
2365|barun sengupta	|director	|personnel	|11/05/47	|7800
5423|n.k.gupta	|chairman	|admin	|30/08/56	|5400
1006|chancah sinhvi	|director	|sales	|03/09/38	|6700
6213|karuna ganguly	|g.m.	|accounts	|05/06/62	|6300
1265|s.n.dasgupta	|manager	|sales	|12/09/63	|5600
4290|jayanth cooudhur	|executive	|production	|07/09/50	|6000
2476|anil agarwal	|manager	|sales	|01/05/59	|5000

```
6521|lalit choudhary   |director     |marketing    |26/09/45    |8200
3212|shyam saksena     |d.g.m.       |accounts     |06/07/47    |7500
2345|j.b. sexena       |g.m.         |marketing    |12/03/45    |8000
0110|v.k.agarwal       |g.m.         |marketing    |31/12/40    |9000
2476|anul aggarwal     |manager      |sales        |01/05/59    |5000
```

grep: Searching for a pattern

The ***grep*** command allows you to search one file or multiple files for lines that contain a pattern

Syntax:- grep [options] pattern [files]

Options

Option	Description
-c	Display the number of matched lines.
-h	Display the matched lines, but do not display the filenames.
-i	Ignore case sensitivity.
-l	Display the filenames, but do not display the matched lines.
-n	Display the matched lines and their line numbers.
-s	Silent mode.
-v	Display all lines that do NOT match.
-e exp	Specifies expression with this option, can use multiple times.
-x	Matches pattern with entire line
-f file	Takes pattern from file, one per file

grep searches for pattern in one or more filenames, or the standard input if no file is specified. The first argument is the pattern and remaining arguments are filenames. Below grep to display the lines containing the string sales from the file emp.lst

```
$grep "sales" emp.lst
2233|a.k.shukla          |g.m.     |sales    |12/12/53|6000
1006|chancah sinhvi      |director |sales    |03/09/38|6700
1265|s.n.deshgupta       |manager  |sales    |12/09/63|5600
2476|anil agarwal        |manager  |sales    |01/05/59|5000
2476|anul aggarwal       |manager  |sales    |01/05/59|5000
```

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

```
who | grep lab >foo
```

```
cat foo
lab    tty1         2013-03-30 05:24 (:0)
lab    pts/0        2013-03-30 05:26 (:0.0)
```

grep can also returns the prompt in case pattern not found.

```
grep rnsit emp.lst                    rnsit not present
$_
```

grep also uses multiple filenames along with the output. Below example searches **from two files called emp.lst and p.lst**

```
grep "director" emp.lst p.lst
emp.lst:9876|jai sharma      |director |production|12/03/50|7000
emp.lst:2365|barun sengutta  |director |personnel |11/05/47|7800
emp.lst:1006|chancah sinhvi  |director |sales     |03/09/38|6700
emp.lst:6521|lalit choudhary |director |marketing |26/09/45|8200
p.lst:2365|barun sengutta    |director |personnel |11/05/47|7800
p.lst:1006|chancah sinhvi    |director |sales     |03/09/38|6700
p.lst:4290|jayanth cooudhur |director|production|07/09/50|6000
p.lst:6521|lalit choudhary   |director |marketing |26/09/45|8200
p.lst:2345|j.b. sexena       |director   |marketing |12/03/45|8000
```

Quoted is necessary when the pattern contains multiple words

```
grep "jai sharma" emp.lst
9876|jai sharma      |director |production|12/03/50|7000
```

grep options Explained:

-i, --ignore-case

Ignore case distinctions in both the *PATTERN* and the input files.

```
grep -i 'Agarwal' emp.lst
2476|anil agarwal    |manager  |sales     |01/05/59|5000
0110|v.k.agarwal     |g.m.     |marketing |31/12/40|9000
```

Deleting Lines (-v)

Display all lines that do NOT match.

```
grep -v 'director' emp.lst >other
```

The file other contains all the lines except director

Displaying Line Number (-n)

Display the matched lines and their line numbers.

```
grep -n 'marketing' emp.lst
```

```
3:5678|sumit chakrabarty|d.g.m.   |marketing |19/04/43|6000
11:6521|lalit choudhary  |director |marketing |26/09/45|8200
13:2345|j.b. sexena      |g.m.     |marketing |12/03/45|8000
14:0110|v.k.agarwal      |g.m.     |marketing |31/12/40|9000
```

Counting Lines containing pattern (-c)

Display the number of matched lines

```
grep -c 'director' emp.lst
```

```
4
```

Displaying file names (-l)

Display the filenames, but do not display the matched lines

```
grep -l 'manager' *.lst
```

```
dest.lst
emp.lst
p.lst
```

Matching multiple patterns (-e)

Specifies expression with this option, can use multiple times.

```
grep –e “Agrawal” –e “aggrawal”  -e  “agrawal” emp.lst
```

Taking pattern from a file (-f)

Takes pattern from file, one per file

```
grep –f pattern.lst emp.lst
```

BASIC REGULAR EXPRESSIONS (BRE)

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the current locale.

A BRE ordinary character, a special character preceded by a backslash or a period matches a single character. A bracket expression matches a single character or a single collating element

Symbol		Matches
^ (Caret)	=	match expression at the start of a line, as in ^A.
$ (Question)	=	match expression at the end of a line, as in A$.
\ (Back Slash)	=	turn off the special meaning of the next character, as in \^.
[ ] (Brackets)	=	match any **one** of the enclosed characters, as in [aeiou]. Use Hyphen "-" for a range, as in [0-9].
[^]	=	match any one character **except** those enclosed in [ ], as in [^0-9].
. (Period)	=	match a single character of any value, except end of line.
* (Asterisk)	=	match zero or more of the preceding character or expression.
\{x,y\}	=	match **x** to **y** occurrences of the preceding.
\{x\}	=	match exactly **x** occurrences of the preceding.
\{x,\}	=	match **x** or more occurrences of the preceding.
Bash$		Bash at the end of a file
^$		Lines containing nothing
^pat		Pattern pat at beginning of a line

The character class

Specify a group of characters enclosed within a pair of rectangular brackets [ ]. Performs single character in the group.

[ra]

Matches either a or r

Example

```
grep "[aA]g[ar][ar]wal" emp.lst
2476|anil agarwal     |manager  |sales     |01/05/59|5000
0110|v.k.agarwal     |g.m.     |marketing |31/12/40|9000
```

Negating a class(^):

All the characters other than this class so [^a-z A-Z] matches other than alphanumeric.

The *:

match zero or more of the preceding character or expression.

Example pattern

g*

matches single character g, or any number of gs

nothing g gg ggg gggg

Example:

```
grep "[aA]gg*[ar][ar]wal" emp.lst
2476|anil agarwal     |manager  |sales     |01/05/59|5000
0110|v.k.agarwal     |g.m.     |marketing |31/12/40|9000
2476|anul aggarwal    |manager  |sales     |01/05/59|5000
```

The Dot

Match a single character of any value, except end of line.

The Regular Expression .*

The dot along with the * (.*) constitutes a very useful regular expression. It signifies any number of characters ,or none.

```
grep "j.*sexena" emp.lst
2345|j.b. sexena      |g.m.     |marketing |12/03/45|8000
```

Specifying Pattern Locations(^ and $):

^ (caret) – For matching at the beginning of a line. $ - For matching at the end of a line.

Example: To extract those lines where the emp-id begins with a 2.

```
grep "^2" emp.lst
```

2233|a.k.shukla |g.m. |sales |12/12/53|6000
2365|barun sengutta |director |personnel |11/05/47|7800
2476|anil agarwal |manager |sales |01/05/59|5000
2345|j.b. sexena |g.m. |marketing |12/03/45|8000

Example:

i)To select those lines where the salary lies between 7000 and 7999,we have to use the $ at the end of the pattern

grep "7...$" emp.lst

9876|jai sharma |director |production|12/03/50|7000
2365|barun sengutta |director |personnel |11/05/47|7800
3212|shyam saksena |d.g.m. |accounts |06/07/47|7500

ii) UNIX has no command that lists only directories. But we can use a pipeline to "grep" those lines from the listing that begin with a d:

$ ls -l| grep "^[^d]"

When Metacharacters Lose Their Meaning

The – loses it's meaning inside the character class if it's not enclosed on either side by a suitable character ,or when placed outside the class.

The . and * lose their meanings when placed inside the character class.

The * is also matched literally if it's the first character of the expression.

Extended Regular Expressions (ERE) and egrep

Extended regular expressions (ERE) make it possible to match dissimilar patterns with a single expression.

The + and ?

+ - Matches one or more occurrences of the previous character.

? - Matches zero or one occurrence of the previoys character.

Example: i) b+ matches b ,bb , bbb, etc.,

ii) b? matches either a single instance of b or nothing.

iii) Using this extended set,we can now have a different regular expression for matching Agarwal and aggarwal.

```
grep -E "[aA]gg?arwal" emp.lst
2476|anil agarwal     |manager  |sales     |01/05/59|5000
0110|v.k.agarwal     |g.m.     |marketing |31/12/40|9000
2476|anul aggarwal    |manager  |sales     |01/05/59|5000
```

Matching Multiple Patterns(| , (and))

The | is the delimiter of multiple patterns. Using it ,we can locate both sengupta and dasgupta from the file and without using the –e option twice:

```
grep -E 'sengupta|dasgupta' emp.lst
2365|barun sengupta   |director |personnel |11/05/47|7800
1265|s.n.dasgupta     |manager  |sales     |12/09/63|5600
```

The characters (and) will group patterns and the use of | inside the parentheses can frame an

even more compact pattern:

```
grep -E '(sen|das)gupta' emp.lst
2365|barun sengupta   |director |personnel |11/05/47|7800
1265|s.n.dasgupta     |manager  |sales     |12/09/63|5600
```

ESSENTIAL SHELL PROGRAMMING

- ✓ Shell is an agency that sits between the user and the UNIX system. Shell is the one which understands all user directives and carries them out. It processes the commands issued by the user. The content is based on a type of shell called Bourne shell.

Shell Scripts

- ✓ When groups of command have to be executed regularly, they should be stored in a file, and the file itself executed as a shell script or a shell program by the user.
- ✓ A shell program runs in interpretive mode. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed. Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts. However the use of extension is not mandatory.
- ✓ Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

Example: script.sh

```
#! /bin/sh                          # script.sh: Sample Shell Script

echo “Welcome to Shell Programming”

echo “Today’s date : `date`”

echo “This months calendar:”

cal `date “+%m 20%y”`               #This month’s calendar.

echo “My Shell :$ SHELL”
```

- ✓ The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell.
- ✓ However, this does not apply to the first line which beings with #. This because, it is an interpreter line which always begins with #! followed by the pathname of the shell to be used for running the script.
- ✓ To run the script we need to first make it executable.
- ✓ This is achieved by using the chmod command as shown below:

```
$ chmod +x script.sh
```

Then invoke the script name as:

$ script.sh

Once this is done, we can see the following output :

Welcome to Shell Programming

Today's date: Mon Oct 8 08:02:45 IST 2007

This month's calendar:

October 2007

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

My Shell: /bin/Sh

As stated above the child shell reads and executes each statement in interpretive mode. We can also explicitly spawn a child of your choice with the script name as argument:

sh script.sh

Read: Making scripts interactive

- ✓ The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user).
- ✓ It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

read name

- ✓ Causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable *name*.

Example: A shell script that uses read to take a search string and filename from the terminal.

```
#! /bin/sh                    # emp1.sh: Interactive version, uses read to accept two inputs
echo "Enter the pattern to be searched: \c"        # No newline read pname
echo "Enter the file to be used: \c"               # use echo –e in bash read fname
```

```
echo "Searching for pattern $pname from the file $fname"

grep $pname $fname

echo "Selected records shown above"
```

Running of the above script by specifying the inputs when the script pauses twice:

```
$ emp1.sh

Enter the pattern to be searched : director

Enter the file to be used: emp.lst

Searching for pattern director from the file emp.lst

        9876  Jai Sharma    Director    Productions
        2356  Rohit         Director    Sales

Selected records shown above
```

Using Command Line Arguments

- ✓ Shell scripts also accept arguments from the command line.
- ✓ Therefore e they can be run non interactively and be used with redirection and pipelines. The arguments are assigned to special shell variables.
- ✓ Represented by $1, $2, etc; similar to C command arguments argv[0], argv[1], etc.
- ✓ The following table lists the different shell parameters.

Shell parameter	**Significance**
$1, $2…	Positional parameters representing command line arguments
$ #	No. of arguments specified in command line
$ 0	Name of the executed command
$ *	Complete set of positional parameters as a single string
"$ @"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	Pid of the current shell
$!	PID of the last background job.

exit and _Exit Status of Command

- ✓ To terminate a program exit is used. Nonzero value indicates an error condition.

Example 1:

$ cat foo

Cat: can't open foo

Returns nonzero exit status. The shell variable $? Stores this status.

Example 2:

grep director emp.lst > /dev/null:echo $?

0

Exit status is used to devise program logic that braches into different paths depending on success or failure of a command.

The logical Operators && and ||

The shell provides two operators that aloe conditional execution, the && and ||.

Usage:

cmd1 && cmd2

cmd1 || cmd2

&& delimits two commands. cmd 2 executed only when cmd1 succeeds.

Example1:

$ grep 'director' emp.lst && echo "Pattern found"

Output:

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Pattern found

Example 2:

$ grep ‘clerk’ emp.lst || echo “Pattern not found”

Output: Pattern not found

Example 3:

grep “$1” $2 || exit 2

echo “Pattern Found Job Over”

The if Conditional

- ✓ The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell

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

If the command succeeds, the statements within if are executed or else statements in else block are executed (if else present).

Example:

```
#! /bin/sh
if grep “^$1” /etc/passwd 2>/dev/null
then
echo “Pattern Found”
else
echo “Pattern Not Found”
fi
```

Output1:

$ emp3.sh ftp

ftp: *.325:15:FTP User:/Users1/home/ftp:/bin/true

Pattern Found

Output2:

$ emp3.sh mail

Pattern Not Found

While: Looping

- ✓ To carry out a set of instruction repeatedly shell offers three features namely while, until and for

Syntax:

while condition is true

do

Commands

Done

The commands enclosed by do and done are executed repeatedly as long as condition is true

Example

```
#! /bin/usr ans=y
while [“$ans”=”y”]
do
echo “Enter the code and description : \c” > /dev/tty
read code description
echo “$code $description” >>newlist
```

```
echo “Enter any more [Y/N]”

read any case $any in

Y* | y* ) answer =y;; N* | n*) answer = n;;

*) answer=y;;

esac

done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

Using test and [ ] to Evaluate Expressions

- ✓ Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement.
- ✓ Test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decisions.
- ✓ Test works in three ways:

1. Compare two numbers
2. Compares two strings or a single one for a null value
3. Checks files attributes

Test doesn't display any output but simply returns a value that sets the parameters $?

Numeric Comparison

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

- ✓ Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.
- ✓ Numeric comparison in the shell is confined to integer values only, decimal values are simply truncated.

Ex:

$x=5;y=7;z=7.2

1. $test $x –eq $y; echo $?

1 *Not equal*

2. $test $x –lt $y; echo $?

0 *True*

3. $test $z –gt $y; echo $?

1 *7.2 is not greater than 7*

2

4. $test $z –eq $y ; echo $y

0 *7.2 is equal to 7*

1

Example 3 and 4 shows that test uses only integer comparison.

- ✓ The script emp.sh uses test in an if-elif-else-fi construct (Form 3) to evaluate the shell parameter $#

```
#!/bin/sh                    #emp.sh: using test, $0 and $# in an if-elif-else-fi construct

If test $# -eq 0; then

Echo "Usage : $0 pattern file" > /dev/tty

Elfi test $# -eq 2 ;then

Grep "$1" $2 || echo "$1 not found in $2">/dev/tty

Else

echo "You didn't enter two arguments" >/dev/tty

fi
```

- ✓ It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.
- ✓ Run the script four times and redirect the output every time

```
$emp31.sh>foo

Usage : emp.sh pattern file

$emp31.sh ftp>foo

You didn't enter two arguments

$emp31.sh henry /etc/passwd>foo

Henry not found in /etc/passwd

$emp31.sh ftp /etc/passwd>foo

ftp:*:325:15:FTP User:/user1/home/ftp:/bin/true
```

Shorthand for test

[and] can be used instead of test. The following two forms are equivalent

```
Test $x –eq $y

and

[ $x –eq $y ]
```

String Comparison

- ✓ Test command is also used for testing strings. Test can be used to compare strings with the following set of comparison operators as listed below

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
stg	String stg is assigned and not null
s1= =s2	String s1=s2

Table: String test used by test

Example:

```
#!/bin/sh
#emp1.sh checks user input for null values finally turns emp.sh developed previously
if [ $# -eq 0 ] ; then
echo “Enter the string to be searched :\c”
read pname
if [ -z “$pname” ] ; then
echo “You have not entered th e string”; exit 1
fi
echo “Enter the filename to be used :\c”
read flname
if [ ! –n “$flname” ] ; then
echo “ You have not entered the flname” ; exit 2
fi
```

emp.sh "$pname" "$flname"

else emp.sh $* fi

Output1:

$emp1.sh

Enter the string to be searched :[Enter] You have not entered the string

Output2:

$emp1.sh

Enter the string to be searched :root

Enter the filename to be searched :/etc/passwd

Root:x:0:1:Super-user:/:/usr/bin/bash

- ✓ When we run the script with arguments emp1.sh bypasses all the above activities and calls emp.sh to perform all validation checks

$emp1.sh jai

You didn't enter two arguments

$emp1.sh jai emp,lst

9878|jai sharma|director|sales|12/03/56|70000

$emp1.sh "jai sharma" emp.lst

- ✓ You didn't enter two arguments Because $* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $* with "$@" (with quote" and then run the script

File Tests

- ✓ Test can be used to test various file attributes like its type (file, directory or symbolic links) or its permission (read, write. Execute, SUID, etc).

Example:

$ ls –l emp.lst

-rw-rw-rw- 1 kumar group 870 jun 8 15:52 emp.lst

$ [-f emp.lst] ; echo $? Ordinary file

0

$ [-x emp.lst] ; echo $? Not an executable.

1

$ [! -w emp.lst] || echo “False that file not writeable”

False that file is not writable

Example: filetest.sh

```
#! /bin/usr
if [! –e $1] : then
Echo “File doesnot exist”
elif [! –r S1]; then
Echo “File not readable”
elif[! –w $1]; then
Echo “File not writable”
Else
Echo “File is both readable and writable”
fi
```

Output:

$ filetest.sh emp3.lst

File does not exist

$ filetest.sh emp.lst

File is both readable and writable

The following table depicts file-related Tests with test

Test	True if
-f file	File exists and is a regular file
-r file	File exists and 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 & Bash Only)
-u file	File exists and has SUID bit set
-k file	File exists and has sticky bit set
-L file	File exists and is a symbolic link (Korn & Bash Only)
f1 –nt f2	File f1 is newer than f2 (Korn & Bash Only)
f1 –ot f2	File f1 is older than f2 (Korn & Bash Only)
f1 –ef f2	File f1 is linked to f2 (Korn & Bash Only)

The case Conditional

- ✓ The case statement is the second conditional offered by the shell. It doesn't have a parallel either in C (Switch is similar) or perl.
- ✓ The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching. case also handles string tests, but in a more efficient manner than if.

Syntax:

```
case expression in

Pattern1) commands1 ;;

Pattern2) commands2::

..

..

Pattern3) commands3::

esac
```

- ✓ Case first matches expression with pattern1.
- ✓ if the match succeeds, then it executes commands1, which may be one or more commands.
- ✓ If the match fails, then pattern2 is matched and so forth. Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

Example:

```
#! /bin/sh
#
echo “          Menu\n
1. List of files\n2. Processes of user\n3. Today’s Date
4. Users of system\n5.Quit\nEnter your option: \c”
read choice
case “$choice” in
1)  ls –l;;
2)  ps –f ;;
3)  date ;;
4)  who ;;
5)  exit ;;
*) echo “Invalid option”
esac
```

Output

```
$ menu.sh
Menu
                    1. List of files
                    2. Processes of user
                    3. Today’s Date
                    4. Users of system
                    5. Quit
                    Enter your option: 3
Mon Oct 8 08:02:45 IST 2014
```

Matching Multiple Patterns

case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N)

Example:

```
Echo “Do you wish to continue? [y/n]: \c”

Read ans

Case “$ans” in

Y | y );;

N | n) exit ;;

esac
```

Wild-Cards: case uses them:

- ✓ case has a superb string matching feature that uses wild-cards.
- ✓ It uses the filename matching metacharacters *, ? and character class (to match only strings and not files in the current directory.)

Example:

```
Case “$ans” in

[Yy] [eE]* );;          Matches YES, yes, Yes, yEs, etc

[Nn] [oO]) exit ;;      Matches no, NO, No, nO

*) echo “Invalid Response”

esac
```

expr: Computation and String Handling

- ✓ The Broune shell uses expr command to perform computations. This command combines the following two functions:
  1. Performs arithmetic operations on integers
  2. Manipulates strings

Computation:

- ✓ expr can perform the four basic arithmetic operations (+, -, *, /), as well as modulus (%) functions. Examples:

```
$ x=3 y=5
$ expr 3+5
8
$ expr $x-$y
-2
$ expr 3 \* 5        Note:\ is used to prevent the shell from interpreting * as metacharacter
15
$ expr $y/$x
1
$ expr 13%5
3
```

- ✓ expr is also used with command substitution to assign a variable.

Example1:

```
$ x=6 y=2 : z=`expr $x+$y`
$ echo $z
8
```

Example2:

```
$ x=5
$ x=`expr $x+1`
$ echo $x
6
```

String Handling:

- ✓ expr is also used to handle strings.
- ✓ For manipulating strings, expr uses two expressions separated by a colon (:). The string to be worked upon is closed on the left of the colon and a regular expression is placed on its right.
- ✓ Depending on the composition of the expression expr can perform the following three functions:

1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

1. Length of the string:

The regular expression .* is used to print the number of characters matching the pattern

Example1:

```
$ expr “abcdefg” : ‘.*’
7
```

Example2:

```
while echo “Enter your name: \c” ;do
read name
if [`expe “$name” :’.*’`    -gt 20] ; then
echo “Name is very long”
 else
break
fi
done
```

2. Extracting a substring:

expr can extract a string enclosed by the escape characters \ (and \).

Example

```
$ st=2007
$ expr “$st” :’..\(..\)’
07                                    Extracts last two characters.
```

3. Locating position of a character:

- ✓ expr can return the location of the first occurrence of a character inside a string.

Example:

```
$ stg = abcdefgh ; expr “$stg” : ‘[^d]*d’
4                                     Extracts the position of character d
```

$0: Calling a Script by Different Names

- ✓ There are a number of UNIX commands that can be used to call a file by different names and doing different things depending on the name by which it is called.
- ✓ $0 can also be to call a script by different names.

Example:

```
#! /bin/sh
lastfile=`ls –t *.c |head -1`
command=$0
exe=`expr $lastfile: '\(.*\).c'`
case $command in
*runc) $exe ;;
*vic) vi $lastfile;;
*comc) cc –o $exe $lastfile &&
Echo "$lastfile compiled successfully";;
esac
```

After this create the following three links:

ln comc.sh

comc ln comc.sh runc

ln comc.sh vic

Output:

$ comc

hello.c compiled successfully.

While: Looping

- ✓ To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Synatx:

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

- ✓ The commands enclosed by do and done are executed repadetedly as long as condition is true.

Example:

```
#! /bin/usr ans=y
while [“$ans”=”y”]
do
echo “Enter the code and description : \c” > /dev/tty
read code description
echo “$code $description” >>newlist
echo “Enter any more [Y/N]”
read any case $any in
Y* | y* ) answer =y;; N* | n*) answer = n;;
*) answer=y;;
esac
Done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

for: Looping with a List

for is also a repetitive structure.

Synatx:

for variable in list

do

commands

done

list here comprises a series of character strings. Each string is assigned to variable specified.

Example:

for file in ch1 ch2; do

> cp $file ${file}.bak

> echo $file copied to $file.bak done

Output:

ch1 copied to ch1.bak

ch2 copied to ch2.bak

Sources of list:

- **List from variables**: Series of variables are evaluated by the shell before executing the loop

Example:

$ for var in $PATH $HOME; do echo "$var" ; done

Output:

/bin:/usr/bin;/home/local/bin;

/home/user1

- **List from command substitution**: Command substitution is used for creating a list. This is used when list is large.

Example:

$ for var in `cat clist`

- **List from wildcards**: Here the shell interprets the wildcards as filenames.

Example:

```
for file in *.htm *.html ; do
sed 's/strong/STRONG/g
s/img src/IMG SRC/g' $file > $$
mv $$ $file
done
```

- **List from positional parameters:**

  Example: emp.sh

```
#! /bin/sh
for pattern in "$@"; do
grep "$pattern" emp.lst || echo "Pattern $pattern not found"
done
```

Output:

$emp.sh 9876 "Rohit"

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

basename: Changing Filename Extensions:

They are useful in chaining the extension of group of files. Basename extracts the base filename from an absolute pathname.

Example1:

```
$basename /home/user1/test.pl
```

Ouput:

```
test.pl
```

set and shift: Manipulating the Positional Parameters

The set statement assigns positional parameters $1, $2 and so on, to its arguments. This is used for picking up individual fields from the output of a program

Example 1:

$ set 9876 2345 6213

$

This assigns the value 9876 to the positional parameters $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $*.

Example 2:

$ set `date`

$ echo $*

Mon Oct 8 08:02:45 IST 2007

Example 3:

$ echo "The date today is $2 $3, $6"

The date today is Oct 8, 2007

Shift: Shifting Arguments Left

Shift transfers the contents of positional parameters to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on

Example 1:

$ echo "$@" *$@ and $* are interchangeable*

Mon Oct 8 08:02:45 IST 2014

$ echo $1 $2 $3

Mon Oct10

$shift

$echo $1 $2 $3

Mon Oct 10 08:02:45

$shift 2 *Shifts 2 places*

$echo $1 $2 $3

08:02:45 IST 2014

Example 2: emp.sh

```
#! /bin/sh
Case $# in
0|1) echo “Usage: $0 file pattern(S)” ;exit ;;
*) fname=$1 shift
for pattern in “$@” ; do
grep “$pattern” $fname || echo “Pattern $pattern not found”
done;;
esac
```

Output:

$emp.sh emp.lst

Insufficient number of arguments

$emp.sh emp.lst Rakesh 1006 9877

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Pattern 9877 not found

Set -- : Helps Command Substitution

In order for the set to interpret - and null output produced by UNIX commands the – option is used. If not used in the output is treated as an option and set will interpret it wrongly. In case of null, all variables are displayed instead of null

Example:

$set `ls –l chp1`

Output:

-rwxr-xr-x: bad options

Example2:

$set `grep usr1 /etc/passwd`

Correction to be made to get correct output are:

$set -- `ls –l chp1`

$set -- `grep usr1 /etc/passwd`

The Here Document (<<)

The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:

mailx kumar << MARK

Your program for printing the invoices has been executed on `date`. Check the print queue

The updated file is $flname

MARK

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.

Using Here Document with Interactive Programs

A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

$ search.sh << END

> director

>emp.lst

>END Output:

Enter the pattern to be searched: Enter the file to be used: Searching for director from file emp.lst

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Selected records shown above.

The script search.sh will run non-interactively and display the lines containing “director”

in the file emp.lst.

trap: interrupting a Program

The shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap ‘command_list’ signal_list

When a script is sent any of the signals in signal_list, trap executes the commands in command list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM

trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list

Example:

trap ‘’ 1 2 15

Simple SHELL Scripts

1)

```
#!/bin/sh

IFS="|"

While echo "enter dept code:\c"; do

read dcode

Set -- `grep "^$dcode"<<limit

01|ISE|22

02|CSE|45

03|ECE|25

04|TCE|58 limit`

Case $# in

3) echo "dept name :$2 \n emp-id:$3\n"

*) echo "invalid code" ;continue

esac done
```

Output:

```
$valcode.sh

Enter dept code:88

Invalid code

Enter dept code:02

Dept name : CSE Emp-id :45

Enter dept code:<ctrl-c>
```

2)

```
#!/bin/sh x=1

While [$x –le 10];do

echo “$x” x=`expr $x+1` done
```

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

for I in “$@” do echo “$I”

sum=`expr $sum + $I`

done

Echo “sum is $sum”
```

3)

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

for i in `cat list`; do echo “string is $I” x= `expr “$I”:’.*’` Echo “length is $x” done
```

4)

This is a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.

For example if A B C are entered then output is C B A.

```
#!/bin/sh

if [ $# -lt 2 ]; then

echo "please enter 2 or more arguments" exit

fi

for x in $@

do

y=$x" "$y done
```

```
echo "$y”
```

Run1:

```
[root@localhost shellprgms]# sh sh1a.sh 1 2 3 4 5 6 7

7 6 5 4 3 2 1
```

5)

The following shell script to accept 2 file names checks if the permission for these files are identical and if they are not identical outputs each filename followed by permission

```
#!/bin/sh

if [ $# -lt 2 ]

then

echo "invalid number of arguments" exit

fi

str1=`ls -l $1|cut -c 2-10`

str2=`ls -l $2|cut -c 2-10`

if [ "$str1" = "$str2" ]

then

echo "the file permissions are the same: $str1" else

echo " Different file permissions "

echo -e "file permission for $1 is $str1\nfile permission for $2 is $str2" fi
```

Run1:

```
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c file permission for ab.c is rw-r--r--

file permission for xy.c is rwxr-xr-x
```

run2:

```
[root@localhost shellprgms]# chmod +x ab.c [root@localhost shellprgms]# sh 2a.sh ab.c xy.c
the file permissions are the same: rwxr-xr-x
```

6)

This shell function that takes a valid directory name as an argument and recursively descends all the subdirectories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

```
#!/bin/sh

if [ $# -gt 2 ]

then

echo "usage sh flname dir" exit

fi

if [ -d $1 ]

then

ls -lR $1|grep -v ^d|cut -c 34-43,56-69|sort -n|tail -1>fn1

 echo "file name is `cut -c 10- fn1`"

echo " the size is `cut -c -9 fn1`"

 else

echo "invalid dir name"

 fi
```

Run1:

```
[root@localhost shellprgms]# sh 3a.sh file name is  a.out

the size is    12172
```

7)

This shell script that accepts valid log-in names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.

```
if [ $# -lt 1 ]

then

echo " Invlaid Arguments..... "
```

```
 exit

fi

for x in “$@”

do

grep -w "^$x" /etc/passwd | cut -d ":" -f 1,6

done
```

Run1:

```
[root@localhost shellprgms]# sh 4a.sh root root:/root
```

Run 2

```
[root@localhost shellprgms]# sh 4a.sh

Invalid Arguments.....
```

8)

This shell script finds and displays all the links of a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not present, the search is to begin in current working directory

```
#!/bin/bash

if [ $# -eq 0 ]

then

echo "Usage:sh 8a.sh[file1] [dir1(optional)]" exit

fi

if [ -f $1 ]

then

dir="." if [ $# -eq 2 ] then

dir=$2

fi
```

```
inode=`ls -i $1|cut -d " " -f 2`
 echo "Hard links of $1 are"
 find $dir -inum $inode –print
echo "Soft links of $1 are"
find $dir -lname $1 -print
else
echo "The file $1 does not exist"
fi
```

Run1:

```
[root@localhost shellprgms]$ sh 5a.sh hai.c
Hard links of hai.c are
./hai.c
Soft links of hai.c are
./hai_soft
```

9)

This shell script displays the calendar for current month with current date replaced by

* or ** depending on whether date has one digit or two digits

```
#!/bin/bash
n=` date +%d`
echo " Today's date is : `date +%d%h%y` ";
cal > calfile if [ $n -gt 9 ] then
sed "s/$n/\**/g" calfile
 else
sed "s/$n/\*/g" calfile
[root@localhost shellprgms]# sh 6a.sh
```

Today's date is : 10 May 14

May 2014

Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7
8	9	**	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

10)

This shell script implements terminal locking. Prompt the user for a password after accepting, prompt for confirmation, if match occurs it must lock and ask for password, if it matches terminal must be unlocked

```
trap “ " 1 2 3 5 20

clear

echo -e “\nenter password to lock terminal:"

 stty -echo

read keynew

stty echo

echo -e “\nconfirm password:"

 stty -echo

read keyold

 stty echo

if [ $keyold = $keynew ]

then


echo "terminal locked!"
```

```
while [ 1 ]
do
echo "retype the password to unlock:"
stty -echo
read key
if [ $key = $keynew ]
then
stty echo
echo "terminal unlocked!"
stty sane
exit
fi
echo "invalid password!"
done
else
echo " passwords do not match!"
fi
stty sane
```

[root@localhost shellprgms]# sh 13.sh enter password:

confirm password:
terminal locked!
retype the password to unlock:
invalid password!
retype the password to unlock:
terminal unlocked!