UNIX Programming (18CS56)

Module 2

(I) ls –l command

- The ***ls*** command is used to list files in a directory, as we have seen earlier.
- The ***–l*** option with the ***ls*** command is used for long listing of files. It displays several attributes of a file, as shown below:

```
akash@akashshegde11:~/Downloads$ ls -l
total 16
-rw-rw-r-- 1 akash akash    0 Oct  1 13:14 12345
-rw-rw-r-- 1 akash akash    0 Oct  1 13:14 ABCDE
-rw-rw-r-- 1 akash akash   44 Oct  1 13:37 chap1
-rw-rw-r-- 1 akash akash   62 Oct  1 13:40 chap2
-rw-rw-r-- 1 akash akash   62 Oct  1 14:11 chap3
drwxrwxr-x 4 akash akash 4096 Oct  1 13:11 dir1
-rw-rw-r-- 1 akash akash    0 Sep 28 11:51 sample.py
```

- The *inode* number is a data structure that stores the details of a file, and the ***ls*** command fetches details of all the files from their respective inodes.
- Each output of ***ls –l*** is preceded by a total count of number of blocks occupied by the files on disk. Each block consists of 512 bytes of data. This number is 1024 bytes of data in Linux systems.
- The file attributes of the ***ls –l*** command gives seven different columns in the output. The description is given by the following table:

Attribute	**Description**
File Type and Permissions	Type and permissions associated with the file. ***d*** (directory), ***b*** or ***c*** (device), ***r*** (read), ***w*** (write), ***x*** (execute)
Links	Number of links associated with the file. Value greater than one – same file has more than one name.
Ownership	Owner of the file – the owner has full authority to change content and permissions.
Group Ownership	Group that owns the file – all the users of a group work on the same file.
File Size	Size of the file in bytes – this value only gives a character count and not the actual occupied disk space.
Last Modification Time	Last modification time of the file – this changes only when the contents of the file change, not permissions/ownership.
Filename	Name of the file in ASCII collating sequence

- ***ls*** command can be used to display only directories in the long listing using ***–d*** option, as shown below:

```
akash@akashshegde11:~/Downloads$ ls -ld */
drwxrwxr-x 2 akash akash 4096 Oct  4 22:37 dir1/
drwxrwxr-x 2 akash akash 4096 Oct  4 20:11 dir2/
drwxrwxr-x 2 akash akash 4096 Oct  5 12:12 dir3/
drwxrwxr-x 2 akash akash 4096 Oct  5 12:01 dir4/
drwxrwxr-x 2 akash akash 4096 Oct  5 12:15 dir5/
drwxrwxr-x 4 akash akash 4096 Oct  5 11:41 sampledir/
drw-rw-r-x 3 akash akash 4096 Oct  8 10:35 scripts_dir/
```

- ***ls*** command can also display the inode numbers of each file in the long listing using ***–i*** option, as shown below:

```
akash@akashshegde11:~/Music$ ls -li
total 20
220096 -rw-rw-r-- 1 akash akash   44 Oct  8 13:34 chap1
220106 -rw-rw-r-- 1 akash akash   62 Oct  8 13:34 chap2
220108 drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir1
220112 drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir2
220113 drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir3
220030 -rw-rw-r-- 1 akash akash    0 Oct  4 22:27 mfile
220088 -rw-rw-r-- 1 akash akash    0 Oct  8 13:03 sample2.py
220080 -rw-rw-r-- 1 akash akash    0 Oct  8 13:03 sample.c
```

(II) File Ownership

- If a user creates a new file, then the user will be the owner of the file.
- If any group can access a user's file, then the user will be the group owner of the file.
- If a user copies another file, then the user will become the owner of the copy.
- A user cannot create files in another user's directories, as the user does not own those directories and does not have write access by default.
- The access privileges of a group are set by the owner of the shared file and not by the members of the group.

- Every user is represented by a *User ID (UID)* in the UNIX file system.
- This UID gives the name and numeric representation of the user.
- It is maintained in the file ***/etc/passwd***
- Every group is represented by a *Group ID (GID)* in the UNIX file system.
- It is maintained in the file ***/etc/group***.
  /etc/passwd also contains the numeric representation of the group.

- The UID and GID associated with the files can be viewed using the id command, as shown below:

```
akash@akashshegde11:~/Music$ id
uid=1000(akash) gid=1000(akash) groups=1000(akash),4(adm)
,24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(
sambashare),999(lxd)
```

(III) File Permissions

- UNIX offers a categorical distribution of file access rights.
- The three categories of users in the UNIX file system are as follows:
  - The owner (user)
  - The group
  - Others (the world)
- Three permissions are accessible by these three categories of users, namely:
  - Read permissions (represented by **r**).
  - Write permissions (represented by **w**).
  - Execute permissions (represented by **x**).
- The owner's (user's) permissions override the group's permissions when one has ownership of the associated file.

(IV) Changing File Permissions – chmod command

- Every file or directory in the UNIX file system is created with a set of default permissions.
- The ***touch*** command is used to create new files in the UNIX file system.
  An example to create a new file using the ***touch*** command is shown below:

```
akash@akashshegde11:~/Music$ ls
chap1  dir1  dir3   sample2.py
chap2  dir2  mfile  sample.c
akash@akashshegde11:~/Music$
akash@akashshegde11:~/Music$ touch newfile
akash@akashshegde11:~/Music$
akash@akashshegde11:~/Music$ ls
chap1  dir1  dir3   newfile     sample.c
chap2  dir2  mfile  sample2.py
```

- ***chmod*** command stands for changing the mode of a file.
- The mode refers to the permissions of the file itself.
- ***chmod*** command is used to set the permissions of one or more files for all three categories of users (user/owner, group and others).

- Note that user and owner terms are used interchangeably and both have the same meaning.
- The ***chmod*** command can only be run by the user (owner) of the file and the superuser (root).
- The chmod command can be used in two ways –
  i. It can be used in a relative manner by specifying the changes to the current permissions of a file.
  ii. It can be used in an absolute manner by specifying the final permissions of a file.

(V) Relative Permissions

- ***chmod*** command changes the permissions specified in the command line, all other permissions are left unchanged.
- The syntax of ***chmod*** command is: ***chmod*** *category operation permission*
- Here, the command line argument is an expression comprising letters and symbols that describe user category and type of permission being assigned or removed.
- The individual components are specified as follows:
  category – the user category (user, group, others)
  operation – the operation to be performed (assign / remove permission)
  permission – the type of permission (read, write, execute)
- Suitable abbreviations for the components are used; compact expressions are framed and then used as argument to ***chmod*** command.
- It is a good practice to check the permissions using ***ls –l*** command before and after using ***chmod***.
- Relative permissions are *assigned* using the + (plus) operator.
  Some examples are shown below.
- Assigning execute permission to the user of *newfile* is given as follows.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod u+x newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rwxrw-r-- 1 akash akash 0 Oct 10 23:35 newfile
```

- Assigning execute permission to user, group and others of *newfile* is given as follows.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod ugo+x newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rwxrwxr-x 1 akash akash 0 Oct 10 23:35 newfile
```

- Assigning execute permission to all users of *newfile* (this is identical to previous one).

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod a+x newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rwxrwxr-x 1 akash akash 0 Oct 10 23:35 newfile
```

- Assigning permissions to multiple files at once is shown below.

```
akash@akashshegde11:~/Music$ ls -l newfile*
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:39 newfile2
-rw-rw-r-- 1 akash akash 0 Oct 10 23:39 newfile3
akash@akashshegde11:~/Music$ chmod o+w newfile newfile2 newfile3
akash@akashshegde11:~/Music$ ls -l newfile*
-rw-rw-rw- 1 akash akash 0 Oct 10 23:35 newfile
-rw-rw-rw- 1 akash akash 0 Oct 10 23:39 newfile2
-rw-rw-rw- 1 akash akash 0 Oct 10 23:39 newfile3
```

- Relative permissions are *removed* using the - (minus) operator.
  An example is shown below.
- Removing execute permission from the user and others of *newfile* is shown below.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-rw- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod uo-w newfile
akash@akashshegde11:~/Music$ ls -l newfile
-r--rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
```

- Multiple permissions can also be set at once.
  An example of assigning write and execute permission to others is given below.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod o+wx newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-rwx 1 akash akash 0 Oct 10 23:35 newfile
```

- Multiple expressions can also be used at once, delimited by commas.
  An example of assigning execute permission to user and others, write permission to others and removing write permission from group is given below.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod o+wx,u+x,g-w newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rwxr--rwx 1 akash akash 0 Oct 10 23:35 newfile
```

- A table is given below which describes all the abbreviations used for the relative permissions.

Abbreviation	Description
Category	**u** – user **g** – group **o** – others **a** – all (**ugo**)
Operation	**+** – assigns permission **-** – removes permission = – assigns absolute permission
Permission	**r** – read permission **w** – write permission **x** – execute permission

(VI) Absolute Permissions

- ***chmod*** command can be used to change absolute permissions without knowing the current permissions of the file.
- The expression passed as the argument consists of a string of three octal numbers (base 8).
- Octal digits range from 0 to 7, thus set of three bits can represent one octal digit.
- The absolute permissions of each category are given below –
  Read permission – octal 4, binary 100
  Write permission – octal 2, binary 010
  Execute permission – octal 1, binary 001
- The numbers for each category are added up and used in ***chmod*** command. For example, read and write permission is 4+2 = 6, all permissions 4+2+1 = 7, and so on.
- A table is given below which describes all the binary and octal representations of permissions used for the absolute permissions.

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

- There are 3 categories and 3 permissions for each category, thus 3 octal digits can describe a file's permissions completely.
- Most significant digit represents the user, whereas the least significant digit represents others.
- An example of assigning absolute read and write permissions to all 3 categories is shown below.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod 666 newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-rw- 1 akash akash 0 Oct 10 23:35 newfile
```

- An example of assigning read and write permissions to user, only read permission to group and others is shown below. These are the default file permissions in UNIX.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-rw-rw- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod 644 newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rw-r--r-- 1 akash akash 0 Oct 10 23:35 newfile
```

- An example of assigning all permissions to all 3 categories is shown below.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rw-r--r-- 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod 777 newfile
akash@akashshegde11:~/Music$ ls -l newfile
-rwxrwxrwx 1 akash akash 0 Oct 10 23:35 newfile
```

- An example of assigning no permissions to all 3 categories is shown below. The file is rendered useless in this case.

```
akash@akashshegde11:~/Music$ ls -l newfile
-rwxrwxrwx 1 akash akash 0 Oct 10 23:35 newfile
akash@akashshegde11:~/Music$ chmod 000 newfile
akash@akashshegde11:~/Music$ ls -l newfile
---------- 1 akash akash 0 Oct 10 23:35 newfile
```

(VII) Recursive Usage of chmod command

- ***chmod*** can descend a directory hierarchy and apply the expression to every file and subdirectory within it using the recursive option (***-R***).

- An example with the ***–R*** option is given below, where all files and subdirectories in the *dir1* directory have execute permission.

```
akash@akashshegde11:~/Music$ ls -ld */
drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir1/
drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir2/
drwxrwxr-x 2 akash akash 4096 Oct  8 13:34 dir3/
akash@akashshegde11:~/Music$ cd dir1
akash@akashshegde11:~/Music/dir1$ ls -l
total 8
-rw-rw-r-- 1 akash akash 44 Oct  8 13:34 chap1
-rw-rw-r-- 1 akash akash 62 Oct  8 13:34 chap2
akash@akashshegde11:~/Music/dir1$ cd ..
akash@akashshegde11:~/Music$ chmod -R a+x dir1
akash@akashshegde11:~/Music$ cd dir1
akash@akashshegde11:~/Music/dir1$ ls -l
total 8
-rwxrwxr-x 1 akash akash 44 Oct  8 13:34 chap1
-rwxrwxr-x 1 akash akash 62 Oct  8 13:34 chap2
```

- Either relative or absolute permission can be used with the ***–R*** option.
- Multiple directories and filenames can also be provided at once. Two such examples are given below.
  chmod -R 755 . (works on hidden files)
  ***chmod -R a+x ** ** (leaves out hidden files)

(VIII) Directory Permissions

- Directories also have their own permissions.
- Permissions of files are directly influenced by the permissions of the directory that is housing them.
- There a few possible cases with respect to directory permissions – a file cannot be accessed even if it has read permission or a file can be changed even if it is write-protected.
- Default directory permissions are ***755*** (***rwxr-xr-x***)
- A directory must never be writable by group and others.
- If files are being changed even though they may appear to be protected, the directory permissions have to be checked for access rights.

(IX) The Shell

- The shell acts as the main interface between user and kernel.
- It looks for special symbols in the command line, performs the tasks associated with them and finally executes the command.
- The shell is unique and multi-faceted –
  - It acts a command interpreter; it interprets and executes commands entered by the user.
  - It is a programming language and allows a user to write shell scripts and execute them.
  - It is also a process that creates an environment for the user to work in.

(X) The Shell's Interpretive Cycle

- The shell follows a cycle in which it performs various tasks. This cycle is described below, point by point.
- It issues the prompt and waits for user to enter a command.
- It then scans the command line for metacharacters after a command is entered and expands abbreviations to recreate a simplified command line.
- Next, it passes on the command line to the kernel for execution.
- It then waits for the command to complete and does not do any work in the meantime.
- Finally, the prompt reappears after command is executed and the shell begins its next cycle.

(XI) Shell Offerings

- There are various shells available in the UNIX system.
- Primarily, there are two families of UNIX shells, namely Bourne and C shell families.
- The Bourne shell family consists of the Bourne shell (***/bin/sh***), the Korn shell (***/bin/ksh***) and the Bash shell (***/bin/bash***).
- The C shell family consists of C shell (***/bin/csh***) and the TC shell (***/bin/tcsh***)

(XII) Wild Cards

- Wild cards are metacharacters that are used to construct the generalized pattern for matching filenames.
- The pattern is framed with ordinary characters (like *chap*) and a metacharacter (like *****) using well-defined rules.
- The pattern is then used as an argument to the command, and the shell will expand it suitably before the command is executed.

- When the file names are similar, the ***** metacharacter can be used to list all the files with similar pattern. Here, all files starting with *chap* are listed using ***** metacharacter.

```
akash@akashshegde11:~/Downloads$ ls chap*
chap1  chap2  chap200  chap25  chap3  chap3500  chap70  chapter
```

- The wild cards and their respective matching descriptions are provided in the table below.

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 that is within the ASCII range of the characters ***x*** and ***z***
[!ijk]	A single character that is not an ***i***, ***j*** or ***k*** (not available in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters ***x*** and ***z*** (not available in C shell)
{pat1,pat2,...}	Patterns ***pat1***, ***pat2***, and so on (not available in Bourne shell)

- The ***** metacharacter matches any number of characters, including none.
- The ***echo*** command can utilize this * to list all files in a directory, similar to the output of ***ls*** command without any options.

```
akash@akashshegde11:~/Downloads/dir1$ echo *
chap1 chap2
```

- The ***** metacharacter can be used before and after the filename to be matched.

```
akash@akashshegde11:~/Downloads$ ls *chap*
11chap50  1chap5  chap1  chap200  chap3     chap70
1chap     2chap   chap2  chap25   chap3500  chapter
```

- The **?** metacharacter matches a single character.
  Here, the files which have exactly one character after *chap* are listed.

```
akash@akashshegde11:~/Downloads$ ls chap?
chap1  chap2  chap3
```

- Here, the files which have exactly two characters after *chap* are listed.

```
akash@akashshegde11:~/Downloads$ ls chap??
chap25  chap70
```

- The **?** metacharacter can also be used before and after the filename to be matched.

```
akash@akashshegde11:~/Downloads$ ls ?chap?
1chap5
```

- The * metacharacter does not match all files beginning with a **.** or /
  Thus, the **.** (dot) must be matched explicitly in a situation where a user wants to list all hidden filenames in a directory.

```
akash@akashshegde11:~/Downloads$ ls .???*
.abcde  .csk  .rcb
```

- The **.** (dot) need not be matched explicitly if the dot is not at the beginning of the filename.

```
akash@akashshegde11:~/Downloads$ ls student*txt
student.txt
```

- The character class comprises a set of characters enclosed by **[** and **]** but matches only a single character within the class.
- For example, ***[125]*** is a character class that matches a single character of ***1***, ***2*** or ***5***.
  Here, the character class has been combined with a string chap and the ls command lists those files that have either ***1***, ***2*** or ***5*** in their filenames.

```
akash@akashshegde11:~/Downloads$ ls chap[125]
chap1  chap2
```

- The range specification in a character class has to be a valid range specification.
  The left side character of the hyphen must have lower ASCII value than right side character of the hyphen.

```
akash@akashshegde11:~/Downloads$ ls chap[2-6]
chap2  chap3
```

- The character class can also be negated using the **!** metacharacter.
  It reverses the matching of a character class.

```
akash@akashshegde11:~/Downloads$ ls chap[!5]
chap1  chap2  chap3
```

- This negation is particularly useful when used to display files of certain extensions.

```
akash@akashshegde11:~/Downloads$ ls *.[!sh]
sample.c
```

- Dissimilar patterns can also be matched using metacharacters. It uses **{ }** for housing the patterns and **,** as the delimiter.
  An example with matching files is given below.

```
akash@akashshegde11:~/Music$ ls *.{c,py}
sample2.py  sample.c
```

- It can also be used for matching directories.

```
akash@akashshegde11:~/Music$ ls {dir1,dir2,dir3}*
dir1:
chap1  chap2

dir2:

dir3:
```

(XIII) Removing the Meaning of Metacharacters

- When a filename consists of metacharacters, such a situation is difficult to deal with.
- It becomes dangerous to execute normal pattern matching commands in such situations.
- That is why all metacharacters must be protected so that the shell does not interpret them.
- There are two solutions for this:
  - Escaping – providing a \ (backslash) before the wild card.
  - Quoting – enclosing the wild card or entire pattern within quotes.

- ***Escaping*** refers to the method of using \ immediately before a metacharacter to remove its special meaning.
- For example ***** means that the asterisk must be matched literally instead of being interpreted as a metacharacter by the shell.
- An example of listing a file named *file** is shown below.

```
akash@akashshegde11:~/Downloads$ ls file1\*
file1*
```

- An example of listing a file named *file[1-3]* is shown below.

```
akash@akashshegde11:~/Downloads$ ls file\[1-3\]
file[1-3]
```

- To escape a space metacharacter, \ has to be used before the space.

```
akash@akashshegde11:~/Downloads$ ls new\ document.txt
new document.txt
```

- To escape the \ itself, another \ has to be used before the \

```
akash@akashshegde11:~/Downloads$ echo \\
\
```

- To escape the newline character and to write long commands, the \ can be used at the end of each line. The secondary prompt > shows up when there are multiple lines.

```
akash@akashshegde11:~/Downloads$ echo abc\
> def\
> ghi
abcdefghi
```

- ***Quoting*** refers to enclosing command line arguments in quotes to remove the special meaning.
- Quoting is helpful for situations consisting of a large number of command line arguments.
- Single quotes protect all special characters, except ' ' (single quote)
- Double quotes protect all special characters, except " " (double quote), **$** and ` (backquote)
- An example with quoting the \ metacharacter is shown below.

```
akash@akashshegde11:~/Downloads$ echo '\'
\
```

- An example with the double quotes is also shown below.

```
akash@akashshegde11:~/Downloads$ ls "new document.txt"
new document.txt
```

(XIV) Redirection – The Three Standard Files

- The shell associates three files with the terminal – two for display and one for input.
- *Standard input* is a file (or stream) representing input, which is connected to the keyboard.
- *Standard output* is a file (or stream) representing output, which is connected to the display.
- *Standard error* is a file (or stream) representing error messages that emanate from the command or shell. It is also connected to the display.
- Three standard files represent three different streams of characters.
- A stream is just sequence of bytes.
- Every command that uses streams will always find these files open and available.
- These files are closed when command completes execution.
- The shell can associate and disassociate each of these files with physical devices or disks.

- ***Standard Input*** – commands read from the standard input file.
- It represents three input sources, namely:
  - The keyboard, the default source.
  - A file using redirection with the < symbol (metacharacter).
  - Another program using a pipeline.
- ***cat*** and ***wc*** without arguments read the file representing standard input.
  The end of input is done by *Ctrl+D*.

```
akash@akashshegde11:~/Downloads$ wc
abc
def
      2       2       8
```

- Redirecting standard input to originate from a file on disk is done using the < symbol and the process is as follows:
  - Shell opens disk file *chap2* for reading
  - Shell unplugs standard input from its default source and assigns it to *chap2*.
  - ***wc*** reads from standard input which has been reassigned by the shell to *chap2*

```
akash@akashshegde11:~/Downloads$ wc < chap2
 1  1 11
```

- When taking input both from file and standard input, the - (hyphen) symbol is used to indicate sequence of taking input when command is taking input from multiple inputs at once.
  For example, ***cat file1 – file2***

- ***Standard Output*** – commands write to standard output file.
- It represents three output destinations, namely:
  - The terminal, default destination.
  - A file using the redirection symbols > and >>
  - As input to another program using a pipeline.
- Writing to a file can be done using > symbol

```
akash@akashshegde11:~/Downloads$ cat chap2 > chap1
akash@akashshegde11:~/Downloads$ cat chap1
abcdefghij
```

- Appending to a file can be done using >> symbol

```
akash@akashshegde11:~/Downloads$ cat chap3 >> chap1
akash@akashshegde11:~/Downloads$ cat chap1
abcdefghij
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

- Redirecting standard output to any file on the disk is done using the > symbol and the process is as follows:
  - Shell opens disk file *chap1* for writing
  - Shell unplugs standard output from its default destination and assigns it to *chap1*.
  - ***cat*** opens the file chap2 for reading.
  - ***cat*** writes to standard output which has been reassigned by the shell to *chap1*
- Removing the contents of a file without opening/deleting it can be done using the following command – ***cat newfile > newfile***
- Each of the three standard files is represented by a number called a file descriptor.
- A file is opened by referring to its pathname.
- Subsequent read and write operations identify the file by this file descriptor.
- The kernel maintains a table of file descriptors for every process running in the system.
- The first three slots are meant for – **0** (standard input), **1** (standard output) and **2** (standard error).
- Descriptors are implicitly prefixed to the redirection symbols, which means that < and **0**< are identical, whereas > and **1**> are also identical.

```
akash@akashshegde11:~/Downloads$ wc 0< chap2
 1  1 11
akash@akashshegde11:~/Downloads$ cat chap3 1> randomfile
akash@akashshegde11:~/Downloads$ cat randomfile
11 11 44 chap1
-rw-rw-r-- 1 akash akash 44 Oct  1 13:37 chap1
```

- ***Standard Error*** – it represents the diagnostic error messages on the screen due to incorrect command or opening a non-existent file.
- Default destination for this file is the terminal.
- It uses **2**> symbol for redirection and **2**>> for appending the error messages.

```
akash@akashshegde11:~/Downloads$ cat asjfas
cat: asjfas: No such file or directory
akash@akashshegde11:~/Downloads$ cat asjfas 2> error_logger
akash@akashshegde11:~/Downloads$ cat asdio
cat: asdio: No such file or directory
akash@akashshegde11:~/Downloads$ cat asdio 2>> error_logger
akash@akashshegde11:~/Downloads$ cat error_logger
cat: asjfas: No such file or directory
cat: asdio: No such file or directory
```

(XV) Connecting Commands – Pipes and Filters

- Two individual streams of data that can be manipulated by shell are the standard input and standard output.
- The connection of these streams of data is done by using *pipes* and *filters*.
- This concept is closely connected to redirection.
- The approach involves one command taking input from another.
- These connecting commands are mainly used to solve text manipulation programs.
- An example is shown below which does not use pipes.

```
akash@akashshegde11:~/Downloads$ ls -l > file_details.txt
akash@akashshegde11:~/Downloads$ wc < file_details.txt
  80  714 4379
```

The process of the above example is as follows:

- The output of ***ls –l*** is redirected to an intermediate file *file_details.txt*
- Then, ***wc*** takes input from *file_details.txt* and displays it.

- The main disadvantages of using such an approach are – the process can be slow for long-running commands, and also the approach compulsorily needs an intermediate file.
- Same example can be done using pipes.

```
akash@akashshegde11:~/Downloads$ ls -l | wc
     79     705    4319
```

The process of the example will now become:

- Output of ***ls -l*** passed directly to the input of ***wc***
- ***ls*** is said to be piped to ***wc***

An interesting thing to note is that the absence of intermediate file itself has reduced the word, line and character counts.

- Thus, the | is known as the pipe operator
- When multiple commands are connected in this way, a *pipeline* is said to be formed.
- The shell sets up this connection and commands have no knowledge of it.
- Another example can be shown wherein re-direction can be used along with pipe.

```
akash@akashshegde11:~/Downloads$ ls -l | wc > store_file_data.txt
akash@akashshegde11:~/Downloads$ cat store_file_data.txt
     80     714    4382
```

The process of this example is as follows:

- Output of ***ls -l*** passed directly to the input of ***wc***
- This is redirected to *store_file_data.txt*

- Multiple pipes can also be used on the same line, as follows.

```
akash@akashshegde11:~/Downloads$ ls | wc | wc
      1       3      24
```

- All programs run simultaneously in a pipeline.
- The shell has a built-in mechanism to control the flow of the stream.
- Both read and write drivers must work in unison.
- When one driver operates faster than another, the appropriate driver must readjust the flow.

  Example: ***ls | more***

```
akash@akashshegde11:~/Downloads$ ls | more
11chap50
1chap
1chap5
2chap
chap1
chap2
chap200
chap25
chap3
chap3500
chap70
chapter
crs_dir
csk
dc
dir1
dir2
dir3
dir4
dir5
--More--
```

  The *Enter* key can be pressed to move the output line by line, until the end of file.
  The *Space* key can be pressed to move the output page by page, until the end of file.
- Command on the left of | must use standard output.
  Command on the right of | must use standard input.
- **Filters** refer to a set of commands that take input from standard input stream, perform operations on them and write output to standard output stream.
- Examples of commands that work as filters: ***head, tail, more, less, cut, paste, sort, pr, tr, uniq, grep, sed, awk***
- These filters mostly work on entire lines or fields in files.
- They have limited functionality when used in standalone mode and are thus, often combined with pipes to perform powerful text manipulation.

(XVI) Regular Expressions (RegEx)

- A string that can be used to describe several sequences of characters is known as a *regular expression*.
- It is used by different UNIX commands such as ***sed, awk, grep***.
- The shell's wild cards are used to match similar filenames with a single expression.
- A regular expression can match a group of similar patterns.

- The metacharacter set for regex overshadows the wild cards used by the shell.
- There are two categories of regular expressions – basic regular expressions (BRE) and extended regular expressions (ERE).

- The ***grep*** command is primarily used to perform the searching and matching operations.
- It stands for *global regular expression print*.
- The ***grep*** command scans its input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs.
- The syntax for ***grep*** command is as follows: ***grep*** *options pattern filenames*
- It searches for *pattern* in one or more *filenames*, or the standard input if no filename is specified.
- A file "*emp.lst*" is used for depicting many ***grep*** commands; it consists of employee details and related information.

```
akash@akashshegde11:~/Downloads$ cat emp.lst
2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
9876 | jai sharma | director | production | 12/03/50 | 7000
5678 | sumit chakrobarty | 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 | chanchal singhvi | 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 | jayant Choudhury | executive | production | 07/09/50 | 6000
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
3212 | shyam saksena | d.g.m. | accounts | 12/12/55 | 6000
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
2345 | j.b. saxena | g.m. | marketing | 12/03/45 | 8000
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

- An example to display lines containing the string "*sales*" in *emp.lst* file is shown below.

```
akash@akashshegde11:~/Downloads$ grep "sales" emp.lst
2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
```

- It can also be used to search standard input for pattern and save the standard output to a file.

```
akash@akashshegde11:~/Downloads$ cat emp.lst | grep sales > file_emp.txt
akash@akashshegde11:~/Downloads$ cat file_emp.txt
2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
```

- Specifying the search strings under quotes is not necessary, but recommended.
- But quoting is essential when the search string contains multiple words or metacharacters.

```
akash@akashshegde11:~/Downloads$ grep "jai sharma" emp.lst
9876 | jai sharma | director | production | 12/03/50 | 7000
```

- Multiple filenames can be used with ***grep***, and the corresponding output will display the filenames where the search strings are present.

```
akash@akashshegde11:~/Downloads$ grep "director" emp.lst emp2.lst
emp.lst:9876 | jai sharma | director | production | 12/03/50 | 7000
emp.lst:2365 | barun sengupta | director | personnel | 11/05/47 | 7800
emp.lst:1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
emp.lst:6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
emp2.lst:9876 | jai sharma | director | production | 12/03/50 | 7000
emp2.lst:2365 | barun sengupta | director | personnel | 11/05/47 | 7800
emp2.lst:1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
emp2.lst:6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
```

- When the ***grep*** command fails, there is nothing displayed on the terminal. The prompt will simply be returned silently.

```
akash@akashshegde11:~/Downloads$ grep president emp.lst
akash@akashshegde11:~/Downloads$
```

- The ***grep*** features several options and these are listed below. The outputs for the options are also displayed after the table.

Option	Significance
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Display list of filenames only
-e exp	Specifies expression **exp**. 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 patterns as extended regular expression (ERE)
-F	Matches multiple fixed strings

-i option

```
akash@akashshegde11:~/Downloads$ grep -i "agarwal" emp.lst
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
```

-v option

```
akash@akashshegde11:~/Downloads$ grep -v "Agarwal" emp.lst
2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
9876 | jai sharma | director | production | 12/03/50 | 7000
5678 | sumit chakrobarty | 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 | chanchal singhvi | 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 | jayant Choudhury | executive | production | 07/09/50 | 6000
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
3212 | shyam saksena | d.g.m. | accounts | 12/12/55 | 6000
2345 | j.b. saxena | g.m. | marketing | 12/03/45 | 8000
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

-n option

```
akash@akashshegde11:~/Downloads$ grep -n "sales" emp.lst
1:2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
6:1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
8:1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
10:2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
```

-c option

```
akash@akashshegde11:~/Downloads$ grep -c "director" emp.lst
4
```

-l option

```
akash@akashshegde11:~/Downloads$ grep -l "manager" *.lst
emp2.lst
emp.lst
```

-e option

```
akash@akashshegde11:~/Downloads$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" em
p.lst
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

(XVII) Basic Regular Expressions (BRE)

- If an expression uses any of the UNIX metacharacter, then it is a part of a regular expression.
- It is primarily used for common query and substitution requirements.
- ***grep*** command supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with ***–E*** option.
- ***sed*** supports only the BRE set.
- The following table gives the various BREs and describes what each BRE matches.

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 p, q or r
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern *pat* at the beginning of the line
pat$	Pattern *pat* at the end of the line
bash$	*bash* at end of line
^bash$	*bash* is the only word in line
^$	Lines containing nothing

- The *character class* is used to match a group of characters enclosed within a pair of rectangular brackets [ ]
- Match is performed for a single character in the group.
  Example is:

```
akash@akashshegde11:~/Downloads$ grep "[aA]g[ar][ar]wal" emp.lst
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

- The * refers to the immediately preceding character.
- Previous character can occur many times, or not at all.
- It is not similar to * used in shell.
  Example is:

```
akash@akashshegde11:~/Downloads$ grep "[aA]gg*[ar][ar]wal" emp.lst
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

- The **.** (dot) – matches a single character.
- It is similar to **?** used in the shell.
- It is extremely powerful when used with *****
  When combined, .* together signifies any number of characters, including none.
  Example is:

```
akash@akashshegde11:~/Downloads$ grep "j.*saxena" emp.lst
2345 | j.b. saxena | g.m. | marketing | 12/03/45 | 8000
```

- The ^ (caret) matches at the beginning of the line.
- Example is:

```
akash@akashshegde11:~/Downloads$ grep "^2" emp.lst
2233 | a.k. shukla | g.m. | sales | 12/12/52 | 6000
2365 | barun sengupta | director | personnel | 11/05/47 | 7800
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
2345 | j.b. saxena | g.m. | marketing | 12/03/45 | 8000
```

- It can also be used for reverse search.
  Example is:

```
akash@akashshegde11:~/Downloads$ grep "^[^2]" emp.lst
9876 | jai sharma | director | production | 12/03/50 | 7000
5678 | sumit chakrobarty | d.g.m. | marketing | 19/04/43 | 6000
5423 | n.k. gupta | chairman | admin | 30/08/56 | 5400
1006 | chanchal singhvi | 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 | jayant Choudhury | executive | production | 07/09/50 | 6000
6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
3212 | shyam saksena | d.g.m. | accounts | 12/12/55 | 6000
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
0110 | v.k. agrawal | g.m. | marketing | 31/12/40 | 9000
```

- The **$** matches at the end of the line.
  Example is:

```
akash@akashshegde11:~/Downloads$ grep "7...$" emp.lst
9876 | jai sharma | director | production | 12/03/50 | 7000
2365 | barun sengupta | director | personnel | 11/05/47 | 7800
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
```

- It is necessary when a pattern occurs in more than one place in a line.
- The ^ has three roles:
  - When it is at the beginning of a character class, it negates every character of the class. *Example*: **[^a-z]**
  - When it is outside the character class, but beginning of the expression, the pattern is matched at the beginning of the line. *Example*: **^2...**
  - When it is present in any other location, it matches itself literally. Example: **a^b**
- Metacharacters lose their meaning when they are used with character classes.

- **-** (hyphen/minus) loses its meaning inside character class if it is not enclosed on either side by a suitable character, or when placed outside the class.
- **.** and ***** also lose their meaning inside character class.
- Thus, escaping necessary for matching metacharacters, such as **\[** or **\.***

(XVIII) Extended Regular Expressions (ERE)

- It is possible to match dissimilar patterns with a single expression. This can be done with extended regular expressions.
- *Usage*: **grep** with the **-E** option, or just **egrep** without any options.
- Two special characters are used in the extended set: **+** and **?**
- The **+** (plus) is used to match one or more occurrences of the previous character.
  Example is shown below.

```
akash@akashshegde11:~/Downloads$ egrep "[aA]gg+arwal" emp.lst
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ grep -E "[aA]gg+arwal" emp.lst
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
```

- The **?** (question mark) is used to match zero or one occurrence of the previous character.
  Example is shown below.

```
akash@akashshegde11:~/Downloads$ grep -E "[aA]gg?arwal" emp.lst
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
3564 | sudhir Agarwal | executive | personnel | 06/07/47 | 7500
```

- Matching multiple patterns can be done using **|** **(** and **)**
  Two examples are shown below.

```
akash@akashshegde11:~/Downloads$ 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
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ 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
```

- A table summarizing the different EREs is given below.

Expression	Matches
ch+	One or more occurrences of character *ch*
ch?	Zero or one occurrence of character *ch*
exp1|exp2	Either *exp1* or *exp2*
(exp1|exp2)exp3	Either *exp1exp3* or *exp2exp3*

(XIX) Variables

- A **variable** is a character string to which a value is assigned.
- The value assigned could be a number, text, filename, device.
- *Syntax*: **variable_name=variable_value**
- *Examples*: **a=2, student1_name="abcde"**
- Variables are accessed using the **$** operator.
  Example:

```
akash@akashshegde11:~/Downloads$ student1_name="abcde"
akash@akashshegde11:~/Downloads$ echo $student1_name
abcde
```

- **Local (ordinary) variable** is a variable that is present within the current instance of the shell.
- It is not available to programs that are started by the shell.
- It is set at the command line and lost when the terminal is shut down.
- It is the default variable available in the command line.
- *Examples of local variables*: **a=2, student1_name="abcde", DOWNLOAD_DIR=/home/abcde/Downloads**
- **Environment variable** is a variable that is available in the user's total environment, i.e., the sub-shells that run shell scripts, mail commands, editors.
- It is available to any child process of the shell.
- *Examples of environment variables*: **HOME, PATH, SHELL**
- The ***set*** command displays all variables available in the current shell. A small part of the output is shown below.

```
akash@akashshegde11:~/Downloads$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_alias
teractive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_LINENO=()
BASH_REMATCH=([0]="\$st" [1]="\$" [2]="st")
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="3" [2]="48" [3]="1" [4]="release"
BASH_VERSION='4.3.48(1)-release'
CLUTTER_IM_MODULE=xim
COLUMNS=108
COMPIZ_CONFIG_PROFILE=ubuntu
COMP_WORDBREAKS=$' \t\n"\'><=;|&(:'
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-3JuXNUTnT4
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
DESKTOP_SESSION=ubuntu
DIRSTACK=()
DISPLAY=:0
EUID=1000
```

- The ***env/printenv*** command displays only environment variables. A small part of the output is shown below.

```
akash@akashshegde11:~/Downloads$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/akash
GPG_AGENT_INFO=/home/akash/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
TERM=xterm-256color
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=52428810
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/6102
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=akash
```

- The following table gives a few important environment variables and their significances.

Environment Variable	**Significance**
HOME	Home directory – the directory a user is placed on logging in
PATH	List of directories searched by shell to locate a command
LOGNAME or USER	Login name of user
MAIL	Absolute pathname of user's mailbox file
MAILCHECK	Mail checking interval for incoming file
TERM	Type of terminal
PWD	Absolute pathname of current directory (Korn and Bash)
CDPATH	List of directories searched by *cd* when used with a non-absolute pathname
PS1 and PS2	Primary and secondary prompt strings
SHELL	User's login shell and one invoked by programs having shell escapes

(XX) .profile

- **.profile** is one of the login scripts which is executed when the user logs in.
- It can have one of three names in Bash - **.profile**, **.bash_profile** or **.bash_login**.
- It contains commands that are meant to be executed only once in a session.
- Furthermore, it also allows customization of operating environment to suit user's requirements.
- Changes must be saved and user must either log out and log in again or execute the script to observe the effects.

- **ls –a** will display one of these scripts in the *home* directory.

```
akash@akashshegde11:~$ cat .profile
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin directories
PATH="$HOME/bin:$HOME/.local/bin:$PATH"
```

(XXI) Introduction to Shell Scripting

- Shell scripts can be written using text editors such as ***nano*** and executed using the ***bash*** command in Bash shells.
- The first two lines in a shell script usually describe the shell in which the script is being run (using the shebang **#!**) and a brief description of the scope of the script.
- A sample script to demonstrate the use of variables is shown below.

```
akash@akashshegde11:~/Downloads$ nano sample_script_var.sh
```

```
  GNU nano 2.5.3

#!/bin/bash
# Demo of variable

var_1="hello_world"
echo $var_1
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_var.sh
hello_world
```

- A sample script to demonstrate the use of command substitution is also shown below.

```
#!/bin/bash
# Performing command substitution

echo -e "Today's date: `date` \n"

echo "This month's calendar: `cal`"

echo "Shell running these commands: $SHELL"
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_cmdsub.sh
Today's date: Tue Nov 10 23:32:58 IST 2020

This month's calendar:    November 2020
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

Shell running these commands: /bin/bash
```

(XXII) Reading User Input

- **read** statement can be used in scripts to take input from the user and make the script interactive.
- It is usually used with one or more variables.
- Input supplied through the standard input is read into these variables.
- A single read statement can be used with one or more variables by providing multiple arguments.
- If the number of arguments is less than the number of variables, the leftover variables are left unassigned.
  However, if the number of arguments is greater than the number of variables, the leftovers are assigned to last variable.
- *Sample script* - take two inputs (*pattern* and *filename*) using **read** and use **grep** to search for *pattern* in the given *file*.

```
#!/bin/bash
# Interactive script using read

echo -e "Enter the pattern to be searched: \c"
read pat_name

echo -e "\nEnter the file to search in: \c"
read file_name

echo -e "\nSearching for $pat_name in $file_name..."
grep "$pat_name" $file_name

echo -e "\nRequested records are displayed above."
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_grep.sh
Enter the pattern to be searched: manager

Enter the file to search in: emp.lst

Searching for manager in emp.lst...
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000

Requested records are displayed above.
```

(XXIII) Command Line Arguments

- Shell scripts accept command line arguments (CLAs) and can run non-interactively.
- The arguments specified within a shell script are known as ***positional parameters***.
- The command itself is represented by **$0**, the first command line argument by **$1**, the second command line argument by **$2**, and so on.
- There are a few positional parameters and they are listed in the table below.

Positional Parameter	**Significance**
$1, $2, …	Positional parameters representing command line arguments
$#	Number of arguments specified in command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as a separate argument
$?	Exit status of last command

$$	PID (process ID) of current shell
$!	PID of the last background job

- *Sample script* – take two command line arguments (*pattern* and *filename*) as inputs and search for the pattern using **grep**.

```
#!/bin/bash
# Non-interactive script using CLAs

echo "The program is: $0"

echo -e "\nThe pattern to be searched is: $1"
#read pat_name

echo -e "\nThe file to be searched in is: $2"
#read file_name

echo -e "\nThe number of arguments passed is: $#"
echo -e "\nThe arguments passed are: $*"

echo -e "\nSearching for $1 in $2..."
grep "$1" $2

echo -e "\nRequested records are displayed above."
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_cla.sh manager emp.lst
The program is: sample_script_cla.sh

The pattern to be searched is: manager

The file to be searched in is: emp.lst

The number of arguments passed is: 2

The arguments passed are: manager emp.lst

Searching for manager in emp.lst...
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000

Requested records are displayed above.
```

(XXIV) exit and Exit Status

- C programs and shell scripts use the same function to terminate a program, the **exit** function.
- This command is generally run with a numeric argument – **0** for success, **1** for failure. (**exit 0** and **exit 1**)
- A command failure is said to occur if **exit** returns a non-zero value.
- **$?** can be used to check exit status of last command.
- Example for 0 exit status (command successful) is shown below.

```
akash@akashshegde11:~/Downloads$ grep manager emp.lst
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$ echo $?
0
```

- Example for non-zero exit status (command failure) is shown below.

```
akash@akashshegde11:~/Downloads$ grep ceo emp.lst
akash@akashshegde11:~/Downloads$ echo $?
1
```

(XXV) Logical Operators for Conditional Execution

- Two logical operators are used for conditional execution - **&&** and ||
- *Typical syntax:* **cmd1 && cmd2**, **cmd1 || cmd2**
- **&&** - command2 is executed only when command1 succeeds.
  An example is shown below.

```
akash@akashshegde11:~$ grep "manager" emp.lst && echo "Pattern found successfully."
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
Pattern found successfully.
```

- || - command2 is executed only when command1 fails.
  An example is shown below.

```
akash@akashshegde11:~$ grep "ceo" emp.lst || echo "Pattern not found!"
Pattern not found!
```

- || usually used with **exit** command to terminate scripts when some command fails.

(XXVI) if Conditional

- The **if** conditional statement allows for two-way decisions depending on the fulfillment of a certain condition.
- There are three different forms of the **if** conditional. These are depicted below.

Form 1:

```
if command is successful
then
        execute commands
else
        execute commands
fi
```

Form 2:

```
if command is successful
then
        execute commands
fi
```

Form 3:

```
if command is succesful
then
        execute commands
elif command is successful
then...
else...
fi
```

- *Sample script* – searching existence of two users in */etc/passwd* using **if**

```
#!/bin/bash
#Using if and else

if grep "^$1" /etc/passwd 2> /dev/null    #search username at the beginning of the line
then
        echo "Pattern found successfully."
else
        echo "Pattern not found!"
fi
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_if_grep.sh mail
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
Pattern found successfully.
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_if_grep.sh ftp
Pattern not found!
```

(XXVII) test Command and its Shorthand

- The **if** conditional cannot be used to directly handle true/false values returned by expressions.
- The **test** command uses operators to evaluate the condition and returns true/false exit status, which is then used by **if** to make decisions.
- It works in three ways – compares numbers, compares strings, checks file's attributes.
- **test** does not display output directly. It is necessary to use the parameter **$?** to check the exit status of the test command.
- An example is given below, where **test** is used with different integer values and one floating point value.

```
akash@akashshegde11:~$ a=3; b=5; c=5.5
akash@akashshegde11:~$ test $a -eq $b; echo $?
1
akash@akashshegde11:~$ test $a -lt $b; echo $?
0
akash@akashshegde11:~$ test $c -gt $b; echo $?
bash: test: 5.5: integer expression expected
2
```

It is worth noting that **test** does not support floating point values.

It is also possible to check for file attributes and permissions using **test**.

```
akash@akashshegde11:~$ test -f emp.lst; echo $?
0
akash@akashshegde11:~$ test ! -w emp.lst; echo "File is writable"
File is writable
```

- The following tables list the operators that are used for numerical comparison, string comparison and file testing using the **test** command.

Operator for Numerical Comparison	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

Operator for String Comparison	**Meaning**
s1 = s2	String *s1* is equal to string *s2*
s1 != s2	String *s1* is not equal to string *s2*
-n *str*	String *s1* is not a null string
-z *str*	String *s1* is a null string
str	String *s1* is assigned and not null
s1 == s2	String *s1* is equal to string *s2* (Korn and Bash only)

Operator for File Tests	**Meaning**
-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
-L *file*	*file* exists and is a symbolic link (Korn and Bash only)
f1 –nt f2	*f1* is newer than *f2* (Korn and Bash only)
f1 –ot f2	*f1* is older than *f2* (Korn and Bash only)
f1 –ef f2	*f1* is linked to *f2* (Korn and Bash only)

- A shorthand is also used for the **test** command and it is widely popular.
  It contains a pair of rectangular brackets enclosing the expression.

```
akash@akashshegde11:~$ a=3; b=5; c=5.5
akash@akashshegde11:~$ test $a -lt $b; echo $?
0
akash@akashshegde11:~$ [ $a -lt $b ]; echo $?
0
```

- *Sample script* – evaluating shell parameter **$#**
  This script uses the positional parameters to search for a pattern in a file, using an if-elif-else-fi construct.

```
#!/bin/bash
# Using test, $0 and $# in an if-elif-else-fi construct

if [ $# -eq 0 ]
then
        echo "Usage: bash $0 pattern file" > /dev/tty
elif [ $# -eq 2 ]
then
        grep "$1" $2 || echo "$1 not found in $2" > /dev/tty
else
        echo "You did not enter two arguments!" > /dev/tty
fi
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_test_numeric.sh
Usage: bash sample_script_test_numeric.sh pattern file
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_numeric.sh ceo emp.lst
ceo not found in emp.lst
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_numeric.sh manager emp.
lst
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_numeric.sh manager
You did not enter two arguments!
```

- *Sample script* – checking for null string in both interactive and non-interactive mode. This script uses the positional parameters to search for a pattern in a file, using an if-elif-else-fi construct.

```
#!/bin/bash
# Checks user input for null strings

if [ $# -eq 0 ] ; then
        echo -e "Enter pattern to be searched: \c"
        read pat_name
        if [ -z "$pat_name" ] ; then
                echo "You have not entered the pattern!" ; exit 1
        fi
        echo -e "Enter filename in which pattern has to be searched: \c"
        read file_name
        if [ ! -n "$file_name" ] ; then
                echo "You have not entered the filename!" ; exit 2
        fi
        bash sample_script_test_numeric.sh "$pat_name" $file_name
else
        bash sample_script_test_numeric.sh "$@"
fi
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh
Enter pattern to be searched: manager
Enter filename in which pattern has to be searched: emp.lst
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh
Enter pattern to be searched: ceo
Enter filename in which pattern has to be searched: emp.lst
ceo not found in emp.lst
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh manager emp.l
st
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh ceo emp.lst
ceo not found in emp.lst
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh "jai sharma"
emp.lst
9876 | jai sharma | director | production | 12/03/50 | 7000
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh
Enter pattern to be searched:
You have not entered the pattern!
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_test_string.sh
Enter pattern to be searched: manager
Enter filename in which pattern has to be searched:
You have not entered the filename!
```

- *Sample script* – performing various file tests, such as checking existence of a file, checking if a file is readable, checking if a file is writable and so on.

```
#!/bin/bash
# Tests file attributes

if [ ! -e $1 ] ; then
        echo "File does not exist!"
elif [ ! -r $1 ] ; then
        echo "File is not readable!"
elif [ ! -w $1 ] ; then
        echo "File is not writable!"
else
        echo "File is both readable and writable."
fi
```

```
akash@akashshegde11:~/Downloads$ ls -l emp.lst
-r--rw-r-- 1 akash akash 956 Oct 17 12:23 emp.lst
akash@akashshegde11:~/Downloads$ bash sample_script_test_files.sh emp.lst
File is not writable!
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ ls -l emp2.lst
-rw-rw-r-- 1 akash akash 953 Oct 17 12:21 emp2.lst
akash@akashshegde11:~/Downloads$ bash sample_script_test_files.sh emp2.lst
File is both readable and writable.
```

- **"$@"** is recommended instead of **$*** to take in multi-word arguments to a shell script, as **"$@"** interprets each quoted argument as a separate argument.

(XXVIII) case Conditional

- The **case** conditional is similar to the **switch** statement in C.
- It matches an expression for more than one alternative and uses a compact construct to permit multi-way branching.
- The general syntax is given as follows:

```
case expression in
        pattern1) commands1 ;;
        pattern2) commands2 ;;
        pattern3) commands3 ;;
        .....
esac
```

- The **case** conditional checks for the value in the expression and matches the corresponding pattern.
- The commands that are matched under a particular pattern are executed one by one, until all the commands have been exhausted.
- *Sample script* – menu to input choice and display corresponding output.

```
#!/bin/bash
# Uses case to offer a 5-item menu

echo -e "\tMENU\n1.List of files\n2.Calendar\n3.Date\n4.Users of system\n5.Quit\nEnter your choice: \c"
read choice

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

```
akash@akashshegde11:~/Downloads$ bash sample_script_case_menu.sh
        MENU
1.List of files
2.Calendar
3.Date
4.Users of system
5.Quit
Enter your choice: 3
Wed Nov 11 10:45:15 IST 2020
akash@akashshegde11:~/Downloads$ bash sample_script_case_menu.sh
        MENU
1.List of files
2.Calendar
3.Date
4.Users of system
5.Quit
Enter your choice: 5
```

- It can also be used to match multiple patterns as shown in the code fragment below.

```
echo -e "Do you wish to continue? (y/n): \c"
read answer
case "$answer" in
        y|Y) ;;
        n|N) exit ;;
esac
```

- And it can also be used to match wild cards as shown in the code fragment below.

```
case "$answer" in
        [yY][eE][sS]*) ;;
        [nN][oO]) exit ;;
                *) echo "Invalid response!"
esac
```

(XXIX) while Loop

- Loops perform a set of instructions repeatedly.
- Shell offers three types of loops – **while**, **until** and **for** loops.
- The **while** loop repeats the instruction set until the control command returns a true exit status.

- The general syntax is as follows:

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

- *Sample script* – accepting ID and description of products in a loop.

```
#!/bin/bash
# Using while to enter ID and description of products

answer=y
while [ "$answer" = "y" ]
do
        echo -e "Enter the ID and description of product: \c" > /dev/tty
        read prod_id prod_desc
        echo "$prod_id|$prod_desc"
        echo -e "Add more products (y/n)? \c" > /dev/tty
        read choice
        case $choice in
                y*|Y*) answer=y ;;
                n*|N*) answer=n ;;
                    *) answer=y ;;
        esac
done > prod_list.txt
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_while_desc.sh
Enter the ID and description of product: 05 apple
Add more products (y/n)? y
Enter the ID and description of product: 06 tomato
Add more products (y/n)? n
akash@akashshegde11:~/Downloads$ cat prod_list.txt
05|apple
06|tomato
```

- It can be used to wait for a file using **sleep** as shown below.

```
while [ ! -r invoice.lst ]
do
        sleep 60
done
alloc.pl
```

- It can also be used to set up infinite loops using **sleep**, as shown below.

```
while true ; do
        df -t
        sleep 300
done &
```

(XXX) for Loop

- The **for** loop does not test any condition, but uses a list instead.
- The general syntax is as follows:

```
for variable in list
do
        commands
done
```

- *Sample script* – Scan a file repeatedly for each argument passed.

```
#!/bin/bash
# Using for loop to scan a file repeatedly for multiple patterns

for pat_name in "$@"
do
       grep "$pat_name" emp.lst || echo "Pattern $pat_name not found!"
done
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_for_scan.sh manager
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_for_scan.sh ceo
Pattern ceo not found!
```

- There a few possible sources of the list that is input to **for** loop.
- A series of variables can be used as a list.

```
for var in $PATH $HOME $MAIL
do
        echo "$var"
done
```

- Command substitution can be used to create the list.

```
for file in `cat clist`
```

- Wild cards can be present in the list and the shell interprets them as filenames.

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

(XXXI) set and shift – Manipulating the Positional Parameters

- **set** is used to assign arguments to positional parameters.
- It is useful when picking up individual fields from the output of a program.
- Values can be assigned to the positional parameters using **set**, as shown.

```
akash@akashshegde11:~$ set 1234 5678 9000
akash@akashshegde11:~$ echo "\$1 is $1"
$1 is 1234
akash@akashshegde11:~$ echo "\$2 is $2"
$2 is 5678
akash@akashshegde11:~$ echo "The $# arguments are: $*"
The 3 arguments are: 1234 5678 9000
```

- Command substitution can also be done using **set**, as shown.

```
akash@akashshegde11:~$ set `date`
akash@akashshegde11:~$ echo $*
Wed Nov 11 12:38:10 IST 2020
akash@akashshegde11:~$ echo "Today's date: $3 $2 $6"
Today's date: 11 Nov 2020
```

- **shift** is used to transfer the contents of a positional parameter to its immediate lower numbered one.
- It is done as many times as the statement is called - **$2** becomes **$1**, **$3** becomes **$2**, and so on.
- An example is shown below.

```
akash@akashshegde11:~$ set `date`
akash@akashshegde11:~$ echo "$@"
Wed Nov 11 12:39:55 IST 2020
akash@akashshegde11:~$ echo $*
Wed Nov 11 12:39:55 IST 2020
akash@akashshegde11:~$
akash@akashshegde11:~$ echo $1 $2 $3
Wed Nov 11
akash@akashshegde11:~$
akash@akashshegde11:~$ shift
akash@akashshegde11:~$ echo $1 $2 $3
Nov 11 12:39:55
akash@akashshegde11:~$
akash@akashshegde11:~$ shift 2
akash@akashshegde11:~$ echo $1 $2 $3
12:39:55 IST 2020
```

- *Sample script* – using **shift** to search for multiple patterns in a file.

```
#!/bin/bash
# Script using shift

case $# in
        0|1) echo "Usage: $0 file pattern"
             exit 2 ;;
          *) flname=$1
             shift
             for pattern in "$@"
             do
                grep "$pattern" $flname || echo "Pattern not found!"
             done ;;
esac
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_shift.sh
Usage: sample_script_shift.sh file pattern
akash@akashshegde11:~/Downloads$ bash sample_script_shift.sh emp.lst
Usage: sample_script_shift.sh file pattern
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_shift.sh emp.lst manager
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
akash@akashshegde11:~/Downloads$
akash@akashshegde11:~/Downloads$ bash sample_script_shift.sh emp.lst ceo
Pattern not found!
```

```
akash@akashshegde11:~/Downloads$ bash sample_script_shift.sh emp.lst manager ceo
 director
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
Pattern not found!
9876 | jai sharma | director | production | 12/03/50 | 7000
2365 | barun sengupta | director | personnel | 11/05/47 | 7800
1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
6521 | lalit chowdury | director | marketing | 26/09/45 | 8200
```

(XXXII) here Document (<<)

- The **here document** (<<) is used when data that must be read by the program is fixed and limited.
- When there is a necessity to read data from the same file containing the script, **here document** (<<) is used.
- It signifies that the data is here itself, rather than in a separate file.
- Any command that uses standard input can also take input from a **here document**.
- It is useful when used with commands that don't accept a filename.

- Contents of the **here document** are interpreted and processed by shell before they are fed as input to a particular command.
- Command substitutions and variables can be used in input via **here document**, but it is not possible to do so in normal standard input.
- An interactive script can be run non-interactively using **here document**.

```
akash@akashshegde11:~/Downloads$ bash sample_script_grep.sh << END
> manager
> emp.lst
> END
Enter the pattern to be searched:
Enter the file to search in:
Searching for manager in emp.lst...
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000

Requested records are displayed above.
```

(XXXIII) trap Statement – Interrupting a Program

- The default way of terminating shell scripts is by pressing the interrupt key (Ctrl+C).
- But this method is not recommended due to lot of temporary files on disk.
- **trap** specifies a set of things to do if the script receives a signal, and can interrupt a program efficiently.
- It is normally placed at the beginning of a shell script and uses two lists – *command list* and *signal list*.

```
trap 'command_list' signal_list
```

- The signal list consists of integer values/names of one or more signals, and the command list consists of commands that have to be executed when the corresponding signals from the signal list are received.
- An example is shown below.

```
trap 'rm $$* ; echo "Program interrupted" ; exit' HUP INT TERM
```

- **trap** can also be used to ignore signals and continue processing by specifying a null command list.

```
trap '' 1 2 15
```