4. ORDINARY AND ENVIRONMENT VARIABLES

A local 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. They are set at command prompt.

4.1 VARIABLE NAMES

- 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.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix Shell variables would have their names in UPPERCASE.
- The following examples are valid variable names −
  VAR_1
  VAR_2
  TOKEN _A

4.2 DEFINING VARIABLES:

- Variables are defined as follows –
- variable_name=variable_value
  For example:
  NAME="Sumitabha Das"

4.3 ACCESSING VARIABLES:

- To access the value stored in a variable, prefix its name with the dollar sign ($) –
- For example, following script would access the value of defined variable NAME and would print it on STDOUT –

```
#!/bin/sh
NAME="Sumitabha Das"
echo $NAME
```

4.4 ENVIRONMENT VARIABLES –

An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

SHELL: points to the shell defined as default.

DISPLAY : Contains the idenifier for the display that X11 programs should use by default.

HOME: Indicates the home directory of the current user; the default argument for the cd built in command

IFS: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.

PATH : Indicates search path for commands.It is a colon separated list of directories in which the shell looks for commands.
PWD: Indicates the current working directory as set by the cd command.
RANDOM: Generates a random integer between 0 and 32767 each time it is referenced.
SHLVL: Increments by one each time an instance of bash is created.
UID: Expands to the numeric user ID of the current user initialized at shell prompt.

- Following is the sample example showing few environment variables −

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo $TERM
xterm
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

PS1(Prompt String one) and PS2 Environment Variables
The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.
For example, if you issued the command −

```
$PS1='=>'
=>
=>
```

Your prompt would become =>.
When you issue a command that is incomplete, the shell will display a secondary prompt and wait for you to complete the command and hit Enter again.
The default secondary prompt is > (the greater than sign), but can be changed by re-defining the **PS2** shell variable −
Following is the example which uses the default secondary prompt −

```
$ echo "this is a
> test"
this is a
test
$
```

```
$PS= '-->'
$ echo "this is a
--> test"
```

4.5 The .profile File

- The file **/etc/profile** is maintained by the system administrator of your UNIX machine and contains shell initialization information required by all users on a system.
- The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes
- The type of terminal you are using

  A list of directories in which to locate commands

  A list of variables effecting look and feel of your terminal.
- You can check your **.profile** available in your home directory. Open it using **vi**editor and check all the variables set for your environment.

4.6 SHELL SCRIPTS

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

Structure of shell script:

```
#!/bin/sh
# script.sh: Sample shell  script
echo   " Todays date: `date`"
echo    "This month calendar"
 cal    `date`   "+%m  20%y"
echo   "My Shell: $SHELL"
```

output:

```
$sh script.sh
Todays date : Mon Nov 7 10:03:42  IST 2016
This month's calendar:
November 2016
```

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			

```
My shell: /bin/sh
```

- Use your vi editor to create the shell script script.sh.
- The script runs three echo commands and shows the use of variable evaluation and command substitution. It also prints the calendar of the current month.
- Note that the # is comment character, that can be placed anywhere in a line; the shell ignores all characters placed on its right.
- However this doesnot apply to the first line, which also begins with a #. This is interpreter line that was mentioned previously.
- It always begins with #! and is followed by pathname of the shell to be used for running the script. However this line specifies the bourne shell.
- To run the script, make it executable first and then invoke the script name

$chmod a+x script.sh
$sh script.sh

- Shell scripts are executed in a separate child shell process and this sub shell need not be of the same type as your login shell.By default child and parent shell belongs to the same type, but you can provide a interpreter line in the first line of the script to specify a different shell for your script.

4.7 read and readonly commands.

read:MAKING SRIPTS INTERACTIVE

- The read statement is the shell internal tool for taking the input from the user ie making scripts interactive.
- It is used with one or more variables. Input is supplied through the standard input is read into these variables.
- When you use statement like:

  read name

  the script pauses at that point to take input from the keyboard. whatever you enter is stored in the variable name. since this is a form of assignment , no $ is used before the name.
- A single read statement can be used with one or more variables to let you enter multiple arguments.

  read pname flname
- The script asks for a pattern to be entered. Input the string director, which is assigned to the variable pname. Next the script asks for the filename enter the string emp.lst which is assigned to the variable flname.
- grep runs with these two variables as arguments

```
#!/bin/sh
#emp1.sh
#
echo "Enter the pattern to be searched : \c"
read pname
echo " Enter the file to be used : \c"
read flname
echo " Searching for $pname from file $flname"
grep "$pname"  $flname
echo "Selected rows shown above"
```

Output:
$sh emp1.sh
Enter the pattern to be searched:**director**
Enter the file to be used: **emp.lst**
Searching for **director** from file emp.lst
101 sharma|**director**|production|12/03/70|7000
102|barun|**director**|marketing|11/06/67|7800
selected rows shown above

4.8 USING COMMAND LINE ARGUMENTS

- When arguments are specified with a shell script they are assigned to certain special variables-positional parameters.
- $*→ store the complete set of positional parameters as a single string.
- $#--> It is set to the number of arguments specified.

- $0-→ holds the command name itself.
- When arguments are specified in this way the first word (the command itself) is assigned to $0, the second word(the first argument) to $1, the third word(the second argument) to $2.

```
#!/bin/sh
#emp2.sh
#
echo "Program:$0
The number of arguments specified is $#
The arguments are $*"
grep "$1" $2
echo "\n job over"
```

Output:

```
$ sh emp2.sh  director  emp.lst
```

Program: **emp2.sh**
The number of arguments specified is:**2**
The arguments are **director emp.lst**
101| sharma|**director**|production|12/03/70|7000
102|barun|**director**|marketing|11/06/67|7800
job over

4.9 SPECIAL PARAMETERS USED BY THE SHELL.

Shell 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 separate argument
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

4.10 Exit and exit status of Command.

- The shells exit command
  exit 0 Used when everything went fine.
  exit 1 Used when something went wrong

Its through the exit command or function that every command returns an exit status to the caller. Further a command is said to return true exit status if it executes successfully and false if its fails.

- THE PARAMETER $? :It stores the exit status of the last command. It has the value 0 if the command succeeds and a non zero value if it fails. This parameter is set by exit's argument If no exit status is specified then $? is set to zero(true).

- Consider two files file1 which exist in current directory and file2 which does not exist

$ ls –l file1; echo $? **/*file1 attributes are listed**
Output :0 /*exit status $?=0, since cmd executed successfully

$ ls –l file2; echo $? **/*error since file2 doesnot exist**
Output: 1 /*exit status $?=1, since cmd execution failed.

4.11 THE LOGICAL OPERATORS && and || - CONDITIONAL EXECUTION

- The shell provides two operators that allow conditional execution.the && and ||.
- The syntax:

cmd1 && cmd2
cmd1 || cmd2

- Consider a file emp.lst

$cat emp.lst
1066| sharma | **director** |sales |03/09/66 | 7000
1098| Kumar |**director**| production|0/08/67 | 8200
1082|sumith| **manager**|marketing|09/09/73| 7090

- The && delimits two commands ; the command cmd2 is executed only when cmd1 succeeds.

$ grep "director" emp.lst && echo "Pattern found in file"
Output:
1066| sharma | **director** |sales |03/09/66 | 7000
1098| Kumar |**director**| production|0/08/67 | 8200
Pattern found in file

- The || operator plays inverse role. The second command is executed only when the first fails.

$grep " deputy manager" emp.lst || echo "Pattern not found"
Output:
Pattern not found /* cmd1 -**deputy manager** is not found in emp.lst. Hence cmd1 fails. Therefore cmd2 "**pattern not found**" executes.

$grep "manager" emp.lst || echo "Pattern not found"
Output
1082|sumith| **manager**|marketing|09/09/73| 709 /*Here cmd1 is executed successfully i,e manager is found ,therefore cmd2 will not be executed.

4.12 CONDITIONAL STATEMENTS:

4.12.1 If 4.12.2 case

4.12.1 The if CONDITIONAL

If command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif command is successful then .. else .. fi

The **if** statement makes two way decision making depending on the fulfillment of a certain condition.

- **If** also requires a **then.**
- It evaluates the success or failure of the command that is specified in its command line. If command succeeds the sequence of the commands following it is executed. If commands fails then the **else** statement is executed
- Every **if** is closed with corresponding with **fi.**

```
#!/bin/sh
a=10
b=20
if [ $a==$b ]
then
echo "a is equal to b"
elif   [ $a –gt  $b  ]
then
echo " a is greater than b"
elif  [ $a  -lt  $b  ]
then
echo " a is lesser than b"
else
echo " None of the conditions met"
fi
```

output:
a is lesser than b

4.12.2 The case CONDITIONAL

- The case statement is the second conditional offered by the shell
- The statement matches an expression for more than one alternative and uses a compact construct to permit multiway branching.

The general syntax is

```
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 on….Each command list is terminated with a pair of semicolons and the entire construct is closed with esac .

```
#!/bin/sh
#menu.sh
# echo "   MENU \n
1.List of files\n   2.Processes of user\n   3.Todays date\n
4.Users of system\n 5.Quit\n
Enter your option:  \"
```

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

To run the program:

$ sh menu.sh

Output:

```
   MENU
1. List of files
2. Processes of user
3. Todays date
4.Users of system
5. Quit
Enter your option : 3
Sun  Nov  6    18:03:06    IST    2016
```

4.12.2.1 Matching multiple patterns:

- case can also specify same action for more than one pattern.
- For example the expression y|Y can be used to match y in both upper and lower case letters.

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

4.12.2.2 Wild cards: case uses them

- case has a string matching feature that uses wild cards.
- It uses the filename matching meta characters *, ? and the character class but only to match strings but not the files in the current directory.

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

4.13 USING test and [] to evaluate the expressions.

Test uses the certain operators to evaluate the condition on its right and returns either true or false exit status which is then used by if for making decision .

Test works in 3 ways:

4.13.1. Compares two numbers (NUMERIC COMPARISION)

4.13.2. compares two strings or a single one for a null value.(STRING COMPARISION)

4.13.3. checks a file attributes. (FILE TEST)

4.13.1. NUMERIC COMPARISION:

The numeric comparison operators used by test are

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

Numeric comparison in the shell is confined to integer values only , decimal values are simply truncated.

$ x=5; y=7; z=7.2
$ test $x –eq $y ; echo $?
Output : 1

$ test $x –lt $y ; echo $?
Output: 0

$ test $z –gt $y ; echo $?
Output: 1

4.13.2. STRING COMPARISION

test can be used to compare strings with yet another set of operators.

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 null string
Stg	String stg is assigned and not null
s1==s2	String s1= s2(Korn and bash only)

Example:

```
#!/bin/sh
a=”abc”
b=”efg”
if [ $a = $b ]
then
echo “a is equal to b”
else
echo “a is not equal to b”
fi
```

output:
a is not equal to b

4.13.3.FILE TESTS

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

Test	True if File
-f file	File exists and is regular file
-r file	File exist and is readable
-w file	File exists and is writable
-x file	File exists and is execuatble
-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)
-L file	File exists and is symbolic link
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)

$ls –l emp.lst
-rw-rw-rw- 1 kumar group 870 Sep 8 15:52 emp.lst

$ [-f emp.lst] ; echo $? // **-f** (emp.lst file exist and its regular file)
0 // Yes
$ [-x emp.lst] ; echo $? // **-x** (emp.lst file is executable or not)
1 //No

4.14 while Looping

- The while statement repeatedly performs a set of instructions until the control command return a true exit status .
- The general syntax is

  while *condition* is true
  do
  commands
  done
- The commands enclosed by do and done are executed repeatedly as long as condition remains true.

4.14.1 Using while to wait for a file

- There are situations when a program needs to read a file that is created by another program, but it has to wait until the file is created.
- The script, monitfile.sh periodically monitors the disk for the existence of the file. And then executes the program once the file has been located.
- It makes use of the external sleep command that makes the script pauses for the duration in seconds as specified in its arguments .
- The loop executes repeatedly as long as the file invoice.lst cannot be read.
- If the file becomes readable the loop is terminated and the program alloc.pl is executed.
- We use the sleep command to check every 60 seconds for the existence of the file.

4.14.2 Setting up an infinite loop

Suppose you as system administrator want to see the free space available on your disk every five minutes

```
while true
do
df –t
sleep 300
done &
```

df reports free space on disk . sleep command is used to hek for every 300 seconds(5 minutes).
& after done runs loop in background

4.15 for : LOOPING WITH A LIST

The shells for loop differs in structure from the ones used in other programming languages.
There is no three part structure.

```
for variables in list
do
commands
done
```

The loop body also uses the keyword do and done. But the additional parameters here are variable and list. Each whitespace separated word in list is assigned to variable and commands are executed until list is executed .

Ex:

```
$for  file  in chap20  chap21  chap22
 do
 cp  $file  {$file}.bak
 echo $file copied to $file.bak
 done
```

Output:
chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak

POSSIBLE SOURCES OF THE LIST

4.15.1 List from variables:

You can use series of variables in the command line. They are evaluated by the shell before executing the loop

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

Output:

```
/bin:/usr/bin:/home/local/bin        /*$var= $PATH
/home/henry                          /*$var=$HOME
```

4.15.2 List from command substitution

The following for command line picks up its list from clist.

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

4.15.3 List from wild cards

When the list consists of wild cards, the shell interprets them as filename.

```
for file in *.htm  *.html
do
gzip $file
done
```

4.15.4 List from positional parameters

for is also used to process positional parameters that are assigned from command line arguments it uses the shell parameter $@ to represent all command line arguments

```
for pattern in "$@"
```

4.15.5 basename: changing filename extensions

- When basename is used with two arguments it strips off the second argument from the first argument

```
$ basename  note.txt  txt
note.                          //txt stripped off
```

- Used to rename filename extensions from txt to doc

```
for file in *.txt
do
leftname= `basename  $file txt `
mv $file  ${{leftname}doc
done
```

- If for picks up note .txt as the first file,
- Leftname stores note.(with dot)
- mv simply adds doc to the extracted string(note.)

4.16 set and shift: MANIPULATING THE POSITIONAL PARAMETERS

- set assigns its argument to positional parameters $1,$2 and so on.

```
$set 989  878  779
$_
```

This assigns the value 989 to the positional parameter $1, 878 to the positional parameter $2 and 779 to $3

Ex:

$echo "\$1 is $1, \$2 is $2, \$3 is $3"
Output: $1 is 989, $2 is 878, $3 is 779
$echo "The $# arguments are $*"
Output: The 3 arguments are 989 878 779

Shift : Shifting Arguments left

Shift transfers the contents of a positional parameter to its immediate lower numbered one.

```
$ set `date'
$echo "$*
```

Output: Wed Nov 9 09:04:30 IST 2016

$shift
$ echo $1 $2 $3 $4 $5
Output: Nov 9 09:04:30 IST 2016

$shift 2
$echo $1 $2 $3
Output: 09:04:30 IST 2016

4.17 The HERE DOCUMENT (<<)

- The shell uses << symbols to read data from the same file containing the script.
- This is referred to as here document , signifying that the data is here rather than in a separate file .
- If the message is short you can have both the command and message in the same script.

```
mail sharma << MARK
Your program for printing the invoices has been executed
on `date`. The updated file is $flname
MARK
```

- The here document symbol(<<) followed by three lines of data and a delimiter (the string MARK)
- The shell treats every line following the command and delimited by MARK as input to the command.
- Sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluated filename.

4.18 trap: INTERRUPTING A PROGRAM

- By default shell scripts terminate whenever the interrupt key is pressed. It may leave a lot of temporary files on disk .
- The trap statement lets you do things you want in case the script receives a signal. The statement is normally placed at the beginning of a 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 of one or more signals.

  trap **'rm $$* ; eho "Program Interrupted" ; exit '** HUP INT TERM
  trap 'cmd_list' sig_list
- trap is a signal handler.
- Here it first removes all the files expanded from $$*, echoes a message and finally terminates the script when the signals SIGHUP(1), SIGINT(2), SIGTERM(15) are sent to the shell process running the script.
- When the interrupt key is pressed it sends the signal number 2.

4.19 FILE SYSTEM and INODES

- Every file system has a directory structure headed by root.
- If you have three file systems on one hard disk then they will have three separate root directories.

- Of these multiple file systems, one of them is considered to be the main one and contains the most of the essential files of the Unix system. This is the **root** file system.
- Every file is associated with a table that contains all that you could possibly need to know about a file except its name and contents. This table is called the **inode** and is accessed by **inode number.**

The inode contains the following attributes of a file

- File type(regular,directory,device etc)
- File permissions
- Number of links
- The UID of the owner
- The GID of the group owner
- File size in bytes
- Date and time of last modification
- Date and time of last access
- Date and time of last change of the inode
- An array of pointers that keep track of all disk blocks used by the file

4.20 LINKS:HARD AND SYMBOLIC(SOFT) LINKS

4.20.1 HARD LINKS: is merely an additional name for an existing file.

Creating hard links:ln command

- A file is linked with the ln command which takes two filenames as arguments.
- The following command links file1 to file2
  $ln file1 file2
- -i option:The –i option to ls shows the inode number of the files

$ls –li file1 file2

Output:

Inode number	permissions	links	owner	group	size	last modified time	filename
29518	-rwxr-xr-x	**2**	Kumar	metal	**915**	May 4 09:58	file1
29518	-rwxr-xr-x	**2**	Kumar	metal	**915**	May 4 09:58	file2

- Original and link file will have same inode number: all files with inode number 29518
- File size is same for original and linked files:915
- The link count here is 2.

Consider another file- file3 is linked to existing file- file2.

- The link count increases to 3

$ln file2 file3
$ls –li file2 file3
Output:

29518	-rwxr-xr-x	**3**	Kumar	metal	**915**	May 4 09:58	file1
29518	-rwxr-xr-x	**3**	Kumar	metal	**915**	May 4 09:58	file2
29518	-rwxr-xr-x	**3**	Kumar	metal	**915**	May 4 09:58	file3

Application of Hard Links:

1. Links provides some protection against accidental deletion especially when they exist in different directories.
2. Because of links we don’t need to maintain two programs as two separate disk files if there is very

little difference between them.
3. lets consider that you have written a number of programs that read a file foo.txt in /HOME/input_files.
Later you reorganized your directory structure and moved foo.txt to /HOME/data.
What happens to all the programs that look for foo.txt at its original location.
Solution:Just link foo.txt to directory input_files

$ln data/foo.txt input_files

LIMITATIONS

- Hard links cannot be used to have two linked filenames in two filesystem.
- Hard links cannot be used to link a directory even within the same file system.

4.20.2 SYMBOLIC LINKS/SOFT LINKS

- Unlike hard link a symbolic link doesnot have the file contents, but simply the pathname of the file that actually has contents.
- Syntax to create symbolic link
  -s option along with ln command is used to create soft link
- Consider two files note and note.sym. Creating soft link is as follows:
  $ln –s note note.sym
  $ls –li note note.sym

9948 -rw-r—r-- **1** kumar group **80** Feb 16 14:52 note
9952 lrwxrwxrwx **1** kumar group **4** Feb 15 15:07 note.sym→note

- The inode number of linked file is 9952 whereas that of original file is 9948
- The file size of the linked file is 4 whereas the original file is 80

HARD LINKS	SOFT LINKS
merely an additional name for an existing file.	name for any file that contains a reference to another file
Original and link file will have same inode number	Inode number of the link file will be different
If original file is deleted, still link file exist	If original file is deleted , link file will not be accessible
Size of hardlink file is same as original file	Size of soft link file is smaller than original file.
It cannot be created across the partitions	It can be created across the partitions

---

4.21. SIMPLE FILTER Commands

a. pr b. Head c. tail d. Cut e. paste f. Sort g.tr

a. pr :PAGINATING FILES

- The pr command prepares file for printing by adding suitable header, footers and formatted text.

$pr filename
$pr dept.lst

Output:

```
Nov  07  08:30  2016  dept.lst  page1
01:accounts:6213
02:admin:5423
03:marketing:6542
04:sales:1008
```

pr adds five lines of margin at the top and bottom.
The header shows the date and time of last modification of the file along with the filename and page number.

pr options

- **-k option**

pr's -k option prints in k columns

- **-d option**

Doublespaces input

- **-n option**

Number lines

- **-o n**

offset lines by n spaces.

Consider a sample file emp.lst

$cat emp.lst

```
2233|a.k shukla          |general manager|sales      |12/12/52|6000
9876|jai sharma          |director       |production|03/06/50|7000
5678|sumit chakrobarty|deputy manager   |marketing|04/09/43|8000
2365|barun sengupta      |director       |personnel|05/11/47|7600
5423|n.k.gupta           |chairman       |admin    |08/07/56|5400
0110|v.k.agarwal         |general manager|accounts |12/03/45|9900
```

b. head: Displaying the beginning of a file

- The head command as the name suggests displays the top of the file.
- When used without an option it displays the first ten lines of the file
  $head emp.lst
- option
  -n option
  $head -n 3 emp.lst
  Output: **/*displays first three lines of file**

```
2233|a.k shukla          |general  manager|sales      |12/12/52|6000
9876|jai sharma          |director        |production|03/06/50|7000
5678|sumit chakrobarty|deputy manager    |marketing|04/09/43|8000
```

c. tail:Displaying the end of a file

- tail command displays the end of file.
- **$tail -n 3 emp.lst**
  Output: **/*displays last three lines of file**

```
2365|barun sengupta     |director          |personnel|05/11/47|7600
5423|n.k.gupta          |chairman          |admin    |08/07/56|5400
0110|v.k.agarwal        |general manager   |accounts |12/03/45|9900
```

d. cut: Slitting a file vertically

1. cutting columns(-c)

- To extract specific columns you need to use -c option with a list of column numbers delimited by a comma.
- Range can also be used using hyphen
- here is an example to extract name and designation from emp.lst
  $cut -c 6-22,24-38 emp.lst

  Output

```
a.k shukla          general manager
jai sharma          director
sumit chakrobarty deputy manager
barun sengupta     director
n.k.gupta           chairman
v.k.agarwal         general manager
```

2.cutting fields(-f)

- cut uses the tab as the default field delimiter.
- Two options need to be used here
  -d for the field delimiter
  -f for the field list

  $cut -d \| -f 2,3 emp.lst
  Output

```
a.k shukla          general manager
jai sharma          director
sumit chakrobarty deputy manager
barun sengupta     director
n.k.gupta           chairman
v.k.agarwal         general manager
```

  $cut -d \| -f 2,3 emp.lst | tee cutlist1

```
a.k shukla          | general manager
jai sharma          |director
sumit chakrobarty |deputy manager
barun sengupta     |director
n.k.gupta           |chairman
v.k.agarwal         |general manager
```

  $cut -d \| -f 1,4-6 emp.lst | tee cutlist2

```
2233|sales         |12/12/52|6000
9876 |production|03/06/50|7000
```

5678|marketing|04/09/43|8000
2365|personnel|05/11/47|7600
5423|admin |08/07/56|5400
0110|accounts |12/03/45|9900

e. paste:Pasting files

- paste command can be used as follows

$paste cutlist1 cutlist2

a.k shukla | general manager 2233|sales |12/12/52|6000
jai sharma |director 9876 |production|03/06/50|7000
sumit chakrobarty |deputy manager 5678|marketing|04/09/43|8000
barun sengupta |director 2365|personnel|05/11/47|7600
n.k.gupta |chairman 5423|admin |08/07/56|5400
v.k.agarwal |general manager 0110|accounts |12/03/45|9900

- paste can also be used with delimiter option -d

$paste -d "|" cutlist1 cutlist2

a.k shukla | general manager | 2233|sales |12/12/52|6000
jai sharma |director | 9876 |production|03/06/50|7000
sumit chakrobarty |deputy manager | 5678|marketing|04/09/43|8000
barun sengupta |director |2365|personnel|05/11/47|7600
n.k.gupta |chairman | 5423|admin |08/07/56|5400
v.k.agarwal |general manager |0110|accounts |12/03/45|9900

- **joining lines(-s)**

consider theaddress book that contains details of three employees with three lines each
anup kumar
anup_k@yahoo.com
24568799
vinod sharma
vinod_sharma@yahoo.com
87764543
barun gupta
barun_gupta@gmail.com
45943890

$paste -s -d "||\n" addressbook
Output:
anup kumar | anup_k@yahoo.com |24568799
vinod sharma | vinod_sharma@yahoo.com|87764543
barun gupta|barun_gupta@gmail.com|45943890

  - The -s option is used for joining three lines
  - the -d (delimiter option) is used here with ||\n (two vertical bar and one newline character)

- The first | is used to separate the first and second fields.
- The second | is used to separate the second and third field
- The third \n (next line character) is used to display next field in new line

f. sort: ORDERING A FILE

- sorting is the ordering of data in ascending or descending sequence.
- The sort command orders a file
- By default entire line is sorted

$sort shortlist

sort options

option	Description
-t char	Uses delimiter char to identify fields
-k n	Sorts on nth field
-k m,n	Starts sorts on mth field and ends sort on nth field
-k m.n	Starts sorts on nth column of mth field
-u	Removes repeated lines
-n	Sorts numerically
-r	Reverses sort order
-m list	Merges sorted files in list
-c	Checks if file sorted
-o filename	Places output in file filename

Consider a sample file named shortlist with employee details

$cat shortlist

2233|a.k shukla| general manager|sales|12/12/52|6000
9876|jai sharma |director|production|03/06/50|7000
5678|sumit chakrobarty|deputy manager|marketing|04/09/43|8000
2365|barun sengupta|director|personnel|05/11/47|7600
5423|n.k.gupta|chairman|admin|08/07/56|5400

- The -t option is used to specify the delimiter

1.Sorting on primary key(-k)

$sort -t "|" -k 2 shortlist

sorting is done on key 2 i,e field 2 is sorted with | as delimiter

output:

2233|**a.k shukla|** general manager|sales|12/12/52|6000
2365|**barun sengupta**|director|personnel|05/11/47|7600
9876|**jai sharma** |director|production|03/06/50|7000
5423|**n.k.gupta**|chairman|admin|08/07/56|5400
5678|**sumit chakrobarty**|deputy manager|marketing|04/09/43|8000

2. Sorting in reverse order

$ sort -t "|" -r -k 2 shortlist

sort order can be reversed with -r option.

Here the sorting is reversed on field 2 with | as delimiter

Output:

5678|**sumit chakrobarty**|deputy manager|marketing|04/09/43|8000

5423|**n.k.gupta**|chairman|admin|08/07/56|5400
9876|**jai sharma** |director|production|03/06/50|7000
2365|**barun sengupta**|director|personnel|05/11/47|7600
2233|**a.k shukla**| general manager|sales|12/12/52|6000

3. Sorting on secondary key(-k m,n)

Sorting can be done on more than one key i,e you can provide a secondary key to sort.
For ex: If the primary key is third field,and the secondary key is second field, then you need to specify for every -k option

$sort -t "|" -k 3,3 -k 2,2 shortlist

Output:

5423|n.k.gupta|**chairman**|admin|08/07/56|5400
5678|sumit chakrobarty|**deputy manager**|marketing|04/09/43|8000
2365|**barun sengupta**|**director**|personnel|05/11/47|7600
9876|**jai sharma** |**director**|production|03/06/50|7000
2233|a.k shukla| **general manager**|sales|12/12/52|6000

4.Sorting on columns(-k m.n)

Sorting can also be done on a specific character position within field to be the beginning of the sort.
For ex:
If you need to sort the file according to the date of birth, then you need to sort on the seventh and eight h columns positions within fifth field. 19/04/43- this is the fifth field

$sort -t "|" -k 5.7, 5.8 shortlist

Output:

5678|sumit chakrobarty|deputy manager|marketing|04/09/**43**|8000
2365|barun sengupta|director|personnel|05/11/**47**|7600
9876|jai sharma |director|production|03/06/**50**|7000
2233|a.k shukla| general manager|sales|12/12/**52**|6000
5423|n.k.gupta|chairman|admin|08/07/**56**|5400

5.Numeric sort(-n)

$sort numfile

Sort a file based on ASCII collating sequence

10
2
27
4

$sort –n numfile

2
4
10
27

6. Removes repeated lines(-u):

The –u option lets you to remove the repeated lines.

$cut –d "|" –f 2,3 emp.lst| sort –u

Output:

a.k shukla general manager
jai sharma director
sumit chakrobarty deputy manager
n.k.gupta chairman

7. Checks if file sorted(-c)

$sort –c shortlist
Output:
$_ File is sorted

8. Merges sorted files in list(-m)

The –m option can merge two or more files that are sorted individually.

$sort –m foo1 foo2 foo3

9. Places output in filename(-o flname)

The output can be redirected to a file using –o option

$ sort –o shortlist shortlist

---

g. Tr-translating characters

- tr translate filter command manipulates individual characters in a line.
- The syntax is
  tr options expression1 expression2 standard input
- tr command takes it input only from standard input, it doesnt take filename as input

$ tr '|/' '~- ' < emp.lst
output
2233~a.k shukla~general manager~sales~21-02-52~6000
9876~jai sharma~director~production~03-05-50~7000
5678~sumit chakrobarty~deputy manager~marketing~14-9-43~8000
It replaces each occurence of |(vertical bar) by ~ (tilde sign)
and each occurence of /(forward slash) by -(hyphen) sign

$ tr '[a-z] '[A-Z]' < emp.lst
output'
2233~a.k shukla~general manager~sales~21-02-52~6000
9876~jai sharma~director~production~03-05-50~7000
5678~sumit chakrobarty~deputy manager~marketing ~14-9-43~8000

<u>tr options</u>

1. Deleting characters: (-d)

To delete characters |/ from the file emp.lst.

$tr -d '|/' < emp.lst
output:

2233	a.k shukla	general manager	sales	210252	6000
9876	jai sharma	director	production	030550	7000
5678	sumit chakrobarty	deputy manager	marketing	14943	8000

2. Compressing Multiple consecutive characters (-s)

Eliminate all redundant spaces

$tr -s ' ' < emp.lst

output:

2233|a.k shukla| general manager|sales|12/12/52|6000
9876|jai sharma |director|production|03/06/50|7000
5678|sumit chakrobarty|deputy manager|marketing|04/09/43|8000

3. Complementing values of expression(-c)

To delete all characters except | and /

$tr -cd '|/' < emp.lst

||||//||///|/////|/|||/////////////|||||/////////////////|||/////////|

4. Using ASCII octal values and Escape sequences

So to have each field in separate line, So | is replaced with newline character(octal value 012)

$tr '|' '012' < emp.lst

2233
a.k.shukla
general manager
sales
12/12/52
6000

4.22. The umask command and default permissions

- When you create files and directories , the permissions assigned to them depend on the system default settings.
- The UNIX system has the following default file and directory permissions.
  rw-rw-rw-(octal 666) for regular files
  rwxrwxrwx(octal 777) for directories.
- To evaluate the current value of mask by using umask command
  $umask
  022
- This is an octal number that has to be subtracted from the system default to obtain the actual default.
- This becomes 644(666-022) for files and 755(777-022) for directories.
- A user can also use this command to set a new default.

4.23 Two special files

/dev/null:

- Especially in shell programming quite often you would like to check whether program runs successfully without seeing its output on the screen.
- You may not want save this file output either.
- You have a special file that simply accepts any stream without growing in size

$cmp foo1 foo2 >/dev/null
$cat /dev/null.
$ /* checks file size its always zero

/dev/null simply incinerates all output written to it. Whether you direct or append, it always remains

zero.This facility is useful in redirecting the error messages away from the terminal so that they don't appear on screen**.**

/dev/tty:stands for controlling the terminal for the current process.For the shell process you are in /dev/tty is the terminal you are now using.

- Consider for instance that kumar is working on terminal /dev/pts/1 and sharma on /dev/pts/2.
- However both can refer to their own terminals with the same filename /dev/tty.
- Thus if kumar issues the command

  who > /dev/tty
- The list of current users is sent to terminal he is currently using /dev/pts/1.
- Similarly sharma an issue the same command to see the output on his terminal /dev/pts/2
- Thus /dev /tty can be accessed independently by several users at same time without conflict.