MODULE 4

Essential Shell Programming

Ordinary Variables:

The shell supports variables that are useful both in the command line and shell scripts.

A variable assignment is of the form

variable=value

No spaces around =.

The evaluation of a variable requires the $ as prefix to the variable name:

```
$ count=5
$ echo $count
5
```

A variable can be assigned the value of another variable:

```
$ total=$count
$ echo $total
5
```

When the shell reads the command line, it interprets any word preceded by a $ as a variable, and replaces the word by the value of the variable.

Variable names begin with a letter but can contain numerals and the _ as the other characters.

Variable names are case sensitive; x and X are two different variables.

Shell variables are not typed. I fact, we don't even have to declare them before we use them.

All shell variables are of the string type, which means even a number 123 is stored as a astring rather than in binary.

All shell variables are initialized to null strings by default.

The Environment Variables:

Environment variables are available in the user's total environment. By convention, environment variable names are defined in uppercase.

Applications often obtain information on the process environment through these environment variables.

Table 10.1 Common Environment Variables

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	Login name of user
USER	Login name of user
MAIL	Absolute pathname of user's mailbox file
MAILCHECK	Mail checking interval for incoming mail
TERM	Type of terminal
PWD	Absolute pathname of current directory (Korn and Bash only)
CDPATH	List of directories searched by cd when used with a nonabsolute pathname
PS1	Primary prompt string
PS2	Secondary prompt string
SHELL	User's login shell and one invoked by programs having shell escapes

The .profile:

Every shell uses at least one startup script in the user's home directory. This script is executed when the user logs in.

The .profile is one such startup script used by the Bourne shell. The script is stored in the user's home directory at the time of user creation. It is really a shell script that is executed by the shell when one logs on to the system.

The profile contains commands that are meant to be executed only once in a session. The .profile lets us customize our operating environment to suit our requirements. This environment remains in effect throughout the login session. Every time we make changes to this file, we should either log out and log in again or use a special command (called dot) to execute the file:

. . profile

Shell Scripts:

A shell program runs in Interpretive Mode.

It cannot be compiled into a separate executable file as a C program is.

Each statement is loaded into memory when it is to be executed.

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.**

Although we can execute virtually any command at the shell prompt, long sets of commands that are going to be executed more than once should be executed using a script file

We normally use the **.sh** extension for shell scripts.

Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

Example: script.sh

```
#! /bin/sh
# script.sh: Sample Shell Script
echo “Welcome to Shell Programming”
echo “Today’s date : `date`”
echo “This months calendar:”
`cal` `date “+%m 20%y”`          #This month’s calendar.
echo “My Shell :$SHELL”
```

The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell.

However, this does not apply to the first line which beings with #. This because, it is an **interpreter line** which always begins with #! followed by the pathname of the shell to be used for running the script. In the above example the first line indicates that we are using a Bourne Shell.

Comments are documentation we add in a script to help us understand it. Comments are identified with the pound sign token (#).It is recommended that each script contain comments at the beginning to identify its name and purpose.

The most important part of a script is its command. We can use any of the commands available in UNIX. However, they will not be executed until we execute the script.

To run the script we need to first make it executable.

This is achieved by using the **chmod** command as shown below:

$ chmod +x script.sh

After the script has been made executable, it is a command and can be executed just like any other command.

We can invoke the above script name as:

$ script.sh

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

Welcome to Shell Programming

Today's date: Mon Oct 8 08:02:45 IST 200

This month's calendar:

October 2007

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

My Shell: /bin/Sh

As stated above the child shell reads and executes each statement in interpretive mode.

We can also explicitly spawn a child of your choice with the script name as argument:

sh script.sh

Note: Here the script neither requires a executable permission nor an interpreter line.

readonly command:

A named constant defines a value that cannot be changed. Although the Korn shell does not have named constants, we can create the same effect by creating a variable, assigning it a value, and then fixing its value with the readonly command.

The format of readonly command is:

readonly variable-list

read: Making scripts interactive

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables.

Inputs supplied with the standard input are read into these variables.

For instance, the use of statement like

read name

causes the script to pause at that point to take input from the keyboard. Whatever is entered by us will be stored in the variable name.

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

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

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

```
$ emp1.sh
Enter the pattern to be searched : director
Enter the file to be used: emp.lst
Searching for pattern director from the file emp.lst
9876 Jai Sharma Director Productions
```

2356 Rohit Director Sales
Selected records shown above

In the above example, the script first 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. We enter the string **emp.lst**, which is assigned to the variable **flname**.
grep then runs with these two variables as arguments.

A single read statement can be used with one or more variables to let us enter multiple arguments:

read pname flname

If the number of arguments supplied is less than the number of variables accepting them, any leftover variables will simply remain unassigned.
However, when the number of arguments exceeds the number of variables, the remaining words are assigned to the last variable.

Using Command Line Arguments

Shell scripts also accept arguments from the command line. Therefore they can be run non interactively and be used with redirection and pipelines.
The arguments are assigned to special shell variables which are known as positional parameters.
The first argument is read by the shell into the parameter $1, the second argument into $2, and so on.
The following is a list of all special variables used by the shell:

Shell parameter	Significance
$1, $2…	**Positional parameters representing command line arguments**
$#	**No. of arguments specified in command line**
$0	**Name of the executed command**

$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	Pid of the current shell
$!	PID of the last background job.

```
#!/bin/sh
# emp2.sh: Non-interactive version  - uses command line arguments
#
echo "Program: $0                  # $0 contains the program name
The number of arguments specified is $#
The arguments are $*"              # All arguments stored in $*
grep "$1" $2
echo "\nJob Over"
```

Fig. 14.3 **emp2.sh**

The above script , runs grep with two positional parameters that are set by the script arguments, director and emp.lst:

```
$ emp2.sh director emp.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp.lst
1006|chanchal singhvi |director |sales     |03/09/38|6700
6521|lalit chowdury   |director |marketing |26/09/45|8200

Job Over
```

When the arguments are specified in this way, the first word (the command itself) is assigned to $0, the second word (the first argument) to $1, and the third word (the second argument) to $2. We can use more positional parameters in this way up to $9.

When we use a multiword string to represent a single command line argument, we must quote it. To look for string chanchal singhvi, we must use

emp2.sh "chanchal singhvi" emp.lst

All assignments to positional and special parameters are made by the shell.

exit and Exit Status of command :

The shell exit command is generally run with a numeric argument:

exit 0	Used when everything went fine
exit 1	Used when something went wrong

We need not place this statement at the end of every shell script because the shell understands when script execution is complete.

It is through the exit command or function that every command returns an exit status to the caller.

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

Example 1:

$ cat foo

Cat: can't open foo

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

Example 2:

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

0 Success

grep manager emp.lst > /dev/null; echo $?

1 Failure in finding pattern

grep manager emp3.lst > /dev/null; echo $?

2 Failure in opening file

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

The logical Operators && and || - Conditional Execution

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

Usage:

cmd1 && cmd2

cmd1 || cmd2

&& delimits two commands. The command cmd2 executed only when cmd1 succeeds.

Example1:

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

Output:

9876 Jai Sharma Director Productions

2356 Rohit Director Sales

Pattern found

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

Example 2:

$ grep 'clerk' emp.lst || echo "Pattern not found"

Output:

Pattern not found

The || can also be used with the **exit** command. We often need to terminate a script when a command fails.

The above script emp2.sh can be modified to include this feature. The following two lines ensure that the program is aborted when the grep command fails and a message is printed if it succeeds.

Example 3:

grep "$1" $2 || exit 2

echo "Pattern Found Job Over"

The if Conditional

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

Form 1

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

Form 2

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

Form 3

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

if also requires a **then**. It evaluates the success or failure of the command that is specified in its "command line".

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

Ex:

```
#!/bin/sh
# emp3.sh : using if and else
#
if grep "$1" $2 > /dev/null
then
        echo "pattern found"
else
        echo "pattern not found"
fi
```

In the first step, grep is first executed and a simple if-else construct the exit status of the grep.

Using test and [ ] to Evaluate Expressions

Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement.

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

Test works in three ways:

- Compare two numbers
- Compares two strings or a single one for a null value
- Checks files attributes

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

Numeric Comparison

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

Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.

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

Ex:

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

1. $test $x –eq $y; echo $?

1	Not equal

2. $test $x –lt $y; echo $?

0	True

3. $test $z –gt $y; echo $?

1	7.2 is not greater than 7

4. $test $z –eq $y ; echo $y

0	7.2 is equal to 7

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

```
#!/bin/sh
# emp3a.sh : Using test, $0 and $# in an if-elif-if construct
#
if test  $#  -eq 0
then
        echo "Usage : $0 pattern file "
elif test  $#  -eq 2
then
        grep "$1" $2 || echo "$1 not found in $2"
else
        echo "you didn't enter two arguments "
fi
```

It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.

Run the script four times and redirect the output every time

$emp3a.sh>foo

Usage : emp3a.sh pattern file

$emp3a.sh ftp>foo

You didn't enter two arguments

$emp3a.sh henry /etc/passwd>foo

henry not found in /etc/passwd

$emp3a.sh ftp /etc/passwd>foo

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

Shorthand for test

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

test $x –eq $y

and

[$x –eq $y]

String Comparison

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

Test	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null
s1= =s2	String s1=s2 (in Korn and Bash only)

```
#!/bin/sh
#emp4.sh
#
if [ $# -eq 0 ]
then
        echo "Enter the string to be searched: "
        read pname
        if [ -z "$pname" ]
        then
                echo "You have not entered the string"
                exit 1
        fi
        echo "Enter the filename to be used : "
        read flname
        if [ ! -n "$flname" ]
        then
                echo "You have not entered the filename "
                exit 2
        fi
        ./emp3a.sh "$pname" "$flname"
else
        ./emp3a.sh $*
fi
```

The above script behaves both interactively and non- interactively.

When run without arguments, it turns interactive and takes two inputs from us. It then runs **emp3a.sh,** the script developed previously, with the supplied inputs as arguments.

When the script runs in the interactive mode, the check for a null string is made with [-z "$pname"] as well as with [! –n "$flname"].

Output1:

```
$emp1.sh
Enter the string to be searched :[Enter]
You have not entered the string
```

Output2:

```
$emp1.sh
Enter the string to be searched :root
Enter the filename to be searched :/etc/passwd
root:x:0:1:Super-user:/:/usr/bin/bash
```

When we run the script with arguments emp4.sh bypasses all the above activities and calls emp3a.sh to perform all validation checks

$emp4.sh jai

You didn't enter two arguments

$emp4.sh jai emp.lst

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

$emp4.sh "jai sharma" emp.lst

You didn't enter two arguments

Because $* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $* with "$@" (with quote) and then run the script.

File Tests

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

The following table depicts file-related Tests with test:

Test	True if
-f file	File exists and is a regular file
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn & Bash Only)
-u file	File exists and has SUID bit set
-k file	File exists and has sticky bit set
-L file	File exists and is a symbolic link (Korn & Bash Only)
f1 –nt f2	File f1 is newer than f2 (Korn & Bash Only)
f1 –ot f2	File f1 is older than f2 (Korn & Bash Only)
f1 –ef f2	File f1 is linked to f2 (Korn & Bash Only)

Table: file-related Tests with test

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

$ [-f emp.lst] ; echo $? → Ordinary file
0

$ [-x emp.lst] ; echo $? → Not an executable.
1

$ [! -w emp.lst] || echo "False that file not writeable"
False that file is not writable.

Example: filetest.sh

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

Output:
$ filetest.sh emp3.lst
File does not exist

$ filetest.sh emp.lst
File is both readable and writable

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.
case also handles string tests, but in a more efficient manner than if.

Syntax:

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

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

Example:

```
#!/bin/sh

echo "          MENU
1. List of files
2. date
3. calendar
4. Users of system
5. Quit to UNIX "

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

Note:
- case can not handle relational and file test, but it matches strings with compact code. It is very effective when the string is fetched by command substitution.

- case can also handle numbers but treats them as strings.

Matching Multiple Patterns:

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

Example:

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

Wild-Cards: case uses them:

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

Example:

```
case "$ans" in
        ;;                              Matches YES, yes, Yes, yEs, etc
        [Nn] [oO]) exit ;;              Matches no, NO, No, nO
        *) echo "Invalid Response"
esac
```

expr: Computation and String Handling

The Bourne shell uses expr command to perform computations. This command combines the following two functions:

- Performs arithmetic operations on integers
- Manipulates strings

Computation:

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

Examples:

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

```
15
$ expr $y/$x
1
$ expr 13%5
3
```

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

Example1:

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

Example2:

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

String Handling:

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

expr can perform the following three functions:
1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

1. Length of the string:

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

Example1:

```
$ expr “abcdefg” : ‘.*’
7
```

Example2:

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

2. Extracting a substring:
expr can extract a string enclosed by the escape characters \ (and \).

Example:
$ st=2007
$ expr "$st" :'..\(..\)'

07 *Extracts last two characters.*

3 Locating position of a character:
expr can return the location of the first occurrence of a character inside a string.

Example:
$ stg = abcdefgh ; expr "$stg" : '[^d]*d'

4 Extracts the p*osition of character d*

While: Looping

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

Syntax:
while condition is true
do
commands
done

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

Example:

```
#!/bin/sh
answer=y
while [ "$answer" = "y" ]
do
        echo "enter the usn and name"
        read usn name
        echo "$ usn | $ name " >> newlist
        echo "Enter any more (y/n)?"
        read anymore
        case $anymore in
                y|Y) answer=y ;;
                n|N) answer=n  exit;;
                 *) answer=y ;;
        esac
done
```

Input:
Enter the usn and name : 4XX16XX444 Alice
Enter any more [Y/N] :y
Enter the code and description : 4XX16XX555 Bob
Enter any more [Y/N] : [Enter]
Enter the code and description : 4XX16XX555 Tom
Enter any more [Y/N] : n
Output:
$ cat newlist
4XX16XX444 | Alice
4XX16XX555 | Bob
4XX16XX555 | Tom

Using while to wait for a File:

```
#! /bin/sh
# monitfile.sh: Waits for a file to be created
#
while [! –r invoice.lst]          #while the file invoice.lst can't be read
do
        sleep 60
done
alloc.pl                          #Execute this program after exiting the loop
```

In the above shell script, the loop executes repreatedly as long as the file invoice.lst can't be read.
If the file becomes readable, the loop is terminated and the program alloc.pl is executed.
This script can be run in the background as shown below:

monitfile.sh &

We used the sleep command to check every 60 seconds for the existence of the file. Sleep id also quite useful in introducing some delay in shell scripts.

Setting up an infinite Loop:

An infinite loop can be implemented by using **true** as a dummy command with **while.**

Ex:

```
while true;
do
        commands
done
```

Until:

The until statement operates with a reverse logic used in while. With until, the loop body is executed as long as the condition remains false.

```
#! /bin/sh
# monitfile.sh: Waits for a file to be created
#
until [–r invoice.lst]          #until the file invoice.lst can't be read
do
        sleep 60
done
alloc.pl                        #Execute this program after exiting the loop
```

for: Looping with a List

for is also a repetitive structure.

Syntax:

for variable in list
do
        commands
done

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

Example:

```
for file in ch1 ch2; do
> cp $file ${file}.bak
> echo $file copied to $file.bak
done
```

Output:
ch1 copied to ch1.bak
ch2 copied to ch2.bak

Sources of list:

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

Example:

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

Output:

/bin:/usr/bin;/home/local/bin;
/home/user1

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

Example:

```
$ for var in `cat clist`
```

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

Example:

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

• **List from positional parameters**: for is also used to process positional parameters that are assigned from command line arguments.

Example: emp.sh

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

Output:

```
$emp.sh 9876 "Rohit"
```

basename: Changing Filename Extensions

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

Example1:

```
$basename test2.doc doc
```

Ouput:

```
test2
```

Example2: Renaming filename extension from .txt to .doc

```
for file in *.txt ; do
        leftname=`basename $file .txt`          Stores left part of filename
        mv $file ${leftname}.doc
done
```

set and shift: Manipulating the Positional Parameters

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

Example 1:
$ set 9876 2345 6213
$

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

Example 2:
$ set `date`
$ echo $*
Mon Oct 8 08:02:45 IST 2007

Example 3:
$ echo "The date today is $2 $3, $6"
The date today is Oct 8, 2007

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

Example 1:
$ echo $*
Mon Oct 8 08:02:45 IST 2007
$ echo $1 $2 $3
Mon Oct 8
$shift
$echo $1 $2 $3
Oct 8 08:02:45
$shift 2 *Shifts 2 places*
$echo $1 $2 $3
08:02:45 IST 2007

Example 2: emp.sh
```
#! /bin/sh
case $# in
0|1) echo "Usage: $0 file pattern(S)" ;exit ;;
*) fname=$1
shift
for pattern in "$@" ; do
grep "$pattern" $fname || echo "Pattern $pattern not found"
```

```
done;;
esac
```

Output:
$emp.sh emp.lst
Insufficient number of arguments
$emp.sh emp.lst Rakesh 1006 9877
Pattern 9877 not found.

Set -- : Helps Command Substitution:

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

Example:
$set `ls –l chp1`

Output:
-rwxr-xr-x: bad options

Example2:
$set `grep usr1 /etc/passwd`
Correction to be made to get correct output are:

$set -- `ls –l chp1`
$set -- `grep usr1 /etc/passwd`

The Here Document (<<)

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

Example:

```
mailx kumar << MARK
Your program for printing the invoices has been executed
on `date`.Check the print queue
The updated file is $flname
MARK
```

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

Using Here Document with Interactive Programs:
A shell script can be made to work non-interactively by supplying inputs through here document.

Example:
$ search.sh

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Output:
Enter the pattern to be searched: Enter the file to be used: Searching for director from file emp.lst
Selected records shown above.
The script search.sh will run non-interactively and display the lines containing "director" in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk.

The trap statement lets us do the things we want to do when a script receives a signal.

The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap 'command_list' signal_list

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

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

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

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

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

Example:

trap '' 1 2 15

File Systems and inodes

The hard disk is split into distinct partitions, with a separate file system in each partition.
Every file system has a directory structure headed by root.

n partitions = n file systems = n separate root directories

All attributes of a file except its name and contents are available in a table – inode (index node), accessed by the inode number.
The inode contains the following attributes of a file:

- File type
- 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

Please note that, neither the name of the file nor the inode number is stored in the inode. It is the directory that stores the inode number along with the filename.
Every file system has a separate portion set aside for storing inodes, where they are laid out in a contiguous manner. This area is accessible only to the kernel. The inode number is actually the position of the inode in this area.
The **ls** command along with **–i** (inode) option tells us the inode number of a file.
To know inode number of a file:

ls -il tulec05

9059 -rw-r--r-- 1 kumar metal 51813 Jan 31 11:15 tulec05

Where, 9059 is the inode number and no other file can have the same inode number in the same file system.

Hard Links

A file can have multiple filenames. When that happens, we sat the file has more than one **link.**

We can access the file by any of its links and all the names provided to a single file have the same inode number.

The link count is displayed in the second column of the listing.

This count is normally 1, but the following files have two links,

```
-rwxr-xr--    2    kumar    metal   163    Jul 13 21:36 backup.sh
-rwxr-xr--    2    kumar    metal   163    Jul 13 21:36 restore.sh
```

All attributes seem to be identical, but the files could still be copies. It's the link count that seems to suggest that the files are linked to each other.

But this can only be confirmed by using the **–i** option to **ls**.

ls -li backup.sh restore.sh

```
478274 -rwxr-xr--    2    kumar    metal    163 Jul 13 21:36 backup.sh
478274 -rwxr-xr--    2    kumar    metal    163 Jul 13 21:36 restore.sh
```

Both the files listed above have the same inode number, so there is only file with a single copy on disk.

The changes made to one link are automatically available in the others.

ln: Creating Hard Links

A file is linked with the **ln** command which takes two filenames as arguments (cp command). The command can create both a hard link and a soft link and has syntax similar to the one used by cp.

The following command links emp.lst with employee:

ln emp.lst employee

The –i option to ls shows that they have the same inode number, meaning that they are actually one end the same file:

ls -li emp.lst employee

29518 -rwxr-xr-x	2	kumar	metal	915 may 4 09:58	emp.lst
29518 -rwxr-xr-x	2	kumar	metal	915 may 4 09:58	employee

The link count, which is normally one for unlinked files, is shown to be two.
We can increase the number of links by adding the third file name emp.dat as:

ln employee emp.dat ; ls -l emp*

29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.dat
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.lst
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 employee

We can link multiple files, but then the destination filename must be a directory.
A file is considered to be completely removed from the file system when its link count drops to zero.
ln returns an error when the destination file exists. Use the –f option to force the removal of the existing link before creation of the new one.

Where to use Hard Links

ln data/ foo.txt input_files

It creates link in directory input_files. With this link available, our existing programs will continue to find foo.txt in the input_files directory.
It is more convenient to do this that modifies all programs to point to the new path.
Links provide some protection against accidental deletion, especially when they exist in different directories.
Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program and to a shell script.

A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called.

We can't have two linked filenames in two file systems and we can't link a directory even within the same file system. This can be solved by using symbolic links (soft links).

Symbolic Links:

Unlike the hard linked, a symbolic link doesn't have the file's contents, but simply provides the pathname of the file that actually has the contents.

$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 16 15:07note.sym ->note

Where, l indicate symbolic link file category. -> indicates note.sym contains the pathname for the filename note. Size of symbolic link is only 4 bytes; it is the length of the pathname of note.

It's important that this time we indeed have two files, and they are not identical.

Removing note.sym won't affect us much because we can easily recreate the link. But if we remove note, we would lose the file containing the data. In that case, note.sym would point to a nonexistent file and become a dangling symbolic link.

Symbolic links can also be used with relative pathnames. Unlike hard links, they can also span multiple file systems and also link directories. If we have to link all filenames in a directory to another directory, it makes sense to simply link the directories.

Like other files, a symbolic link has a separate directory entry with its own inode number. This means that **rm** can remove a symbolic link even if its points to a directory. A symbolic link has an inode number separate from the file that it points to. In most cases, the pathname is stored in the symbolic link and occupies space on disk.

umask: DEFAULT FILE AND DIRECTORY PERMISSIONS

When we create files and directories, the permissions assigned to them depend on the system's default setting.

The UNIX system has the following default permissions for all files and directories.

rw-rw-rw- (octal 666) for regular files

rwxrwxrwx (octal 777) for directories

The default is transformed by subtracting the user mask from it to remove one or more permissions. We can evaluate the current value of the mask by using umask without arguments,

$ umask

022

This becomes 644 (666-022) for ordinary files and 755 (777-022) for directories umask 000. This indicates, we are not subtracting anything and the default permissions will remain unchanged. Note that, changing system wide default permission settings is possible using chmod but not by **umask**

FILTERS

Filters are the commands which accept data from standard input manipulate it and write the results to standard output.

Filters are the central tools of the UNIX tool kit, and each filter performs a simple function. Some commands use delimiter, pipe (|) or colon (:). Many filters work well with delimited fields, and some simply won't work without them.

The piping mechanism allows the standard output of one filter serve as standard input of another. The filters can read data from standard input when used without a filename as argument, and from the file otherwise.

The Simple Database

Several UNIX commands are provided for text editing and shell programming.

(emp.lst) - each line of this file has six fields separated by five delimiters. The details of an employee are stored in one single line. This text file designed in fixed format and containing a personnel database. There are 15 lines, where each field is separated by the delimiter |.

$ 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 | directir | 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. sexena | g.m. | marketing | 12/03/45 | 8000
0110 | v.k.agrawal | g.m.| marketing | 31/12/40 | 9000

head – displaying the beginning of the file

ThIS command displays the top of the file. It displays the first 10 lines of the file, when used without an option.

head emp.lst

We can use the –n option to specify a line count and display, say , the first three lines of file

head -n 3 emp.lst

tail – displaying the end of a file:

This command displays the end of the file. It displays the last 10 lines of the file, when used without an option.

tail emp.lst

The –n option can ne used to specify a line count

tail -n 3 emp.lst

displays the last three lines of the file.

We can also address lines from the beginning of the file instead of the end. The +count option allows to do that, where count represents the line number from where the selection should begin.

tail +11 emp.lst

Will display 11th line onwards

Different options for tail are:

1. ***Monitoring*** **the file growth (-f)**
   Use tail –f when we are running a program that continuously writes to a file, and we want to see how the file is growing. We have to terminate this command with the interrupt key.
   Ex:
   tail –f /oracle/app/oracle/product/8.1/orainst/install.log

2. ***Extracting bytes rather than lines (-c)***

   This options is used to copy the bytes from a file relative to the beginning or end of the file.

   Ex:

   tail -c -512 foo *Copies last 512 bytes from foo*

   tail -c +512 foo *Copies everything after skipping 512 bytes*

cut – slitting a file vertically

Cutting Columns (-c) We can cut by using -c option with a list of column numbers, delimited by a comma (cutting columns).

$ cut –c 6-22, 24-32 shortlist

cut also uses a special form for selecting a column from the beginning and up to the end of a line:

cut -c -3,6-22,28-34,55- shortlist

The expression 55- indicates column number 55 to end of line. Similarly, -3 is the same as 1-3.

Cutting Fields (-f) Most files don't contain fixed length lines, so we have to cut fields rather than columns (cutting fields).

cut uses the tab as the default field delimiter, but can also work with a different delimiter. Two options need to used here:

-d for the field delimiter
-f for the field list
The following command cuts the second and third fields from a sample file:

cut -d \| -f 2,3 shortlist | tee cutlist1

will display the second and third columns of shortlist and saves the output in cutlist1. here | is escaped to prevent it as pipeline character

paste – pasting files

When we cut with cut, it can be pasted back with the paste command, vertically rather than horizontally. We can view two files side by side by pasting them.

$paste cutlist1 cutlist2

We can specify one or more delimiters with -d

$paste -d "|" cutlist1 cutlist2

Where each field will be separated by the delimiter |. Even though paste uses at least two files for concatenating lines, the data for one file can be supplied through the standard input.

Joining lines (-s)

The –s option joins line in the same way **vi**'s **J** command does.Let us consider that the file address book contains the details of three persons

$cat addressbook

paste -s addressbook -to print in one single line

paste -s -d "| | \n" addressbook -are used in a circular manner

sort : ordering a file

Sorting is the ordering of data in ascending or descending sequence. The sort command orders a file and by default, the entire line is sorted.

Ex:

$sort shortlist

sort options:

1. ***Sorting on Primary Key (-k)***

   Ex: To sort on the second field in a file shortlist, we can use the –k option of sort command as:

   $sort –t"|" –k 2 shortlist

2. ***Soring on Secondary key***

   We can sort on more than one key, i.e we can provide a secondary key to sort. I fthe primary key is in the third field and the secondary key is the second field, then we need to specify for –k option , where the sort ends.

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

3. Sorting on Columns

We can also specify a character position within a field to the beginning of sort.

$sort –t"|" –k 5.7,5.8 shortlist

The –k option also uses the form **–k m.n** where n is the character position in the m^{th} field.

4. Numeric Sort (-n)

The –n option is used to sort numbers.

$sort –n numfile

5. Removing repreated lines (-u)

The –u(unique) option lets us remove repeated lines from a file.

Summary of sort options are:

-tchar	uses delimiter char to identify fields
-k	n sorts on nth field
-k m,n	starts sort on mth field and ends sort on nth field
-k m.n	starts sort on nth column of mth field
-u	removes repeated lines
-n	sorts numerically
-r	reverses sort order
-f	folds lowercase to equivalent uppercase
-m	list merges sorted files in list
-c	checks if file is sorted
-o flname	places output in file flname

/dev/null and /dev/tty : Two special files

/dev/null: If we would like to execute a command but don't like to see its contents on the screen, we may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

$ cmp foo1 foo2 >/dev/null

$ cat /dev/null

$_

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

$who >/dev/tty