UNIT-I

UNIT-I

What is Unix :

The UNIX operating system is a set of programs that act as a link between the computer and the user.

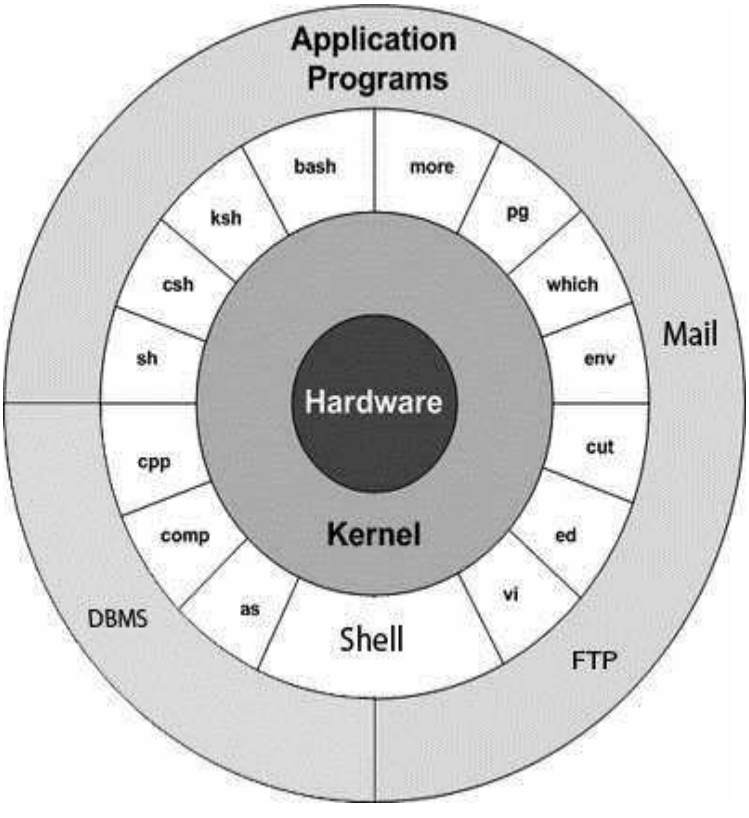
The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel.

Users communicate with the kernel through a program known as the shell. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence UNIX is called multitasking

Unix Architecture:

Here is a basic block diagram of a UNIX system:



The main concept that unites all versions of UNIX is the following four basics:

- **Kernel:** The kernel is the heart of the operating system. It interacts with hardware and most of the tasks like memory management, tash scheduling and file management.
- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all
  commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.
- **Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.
- **Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem

Accessing Unix:

- When you first connect to a UNIX system, you usually see a prompt such as the following

To log in:

1. Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
2. Type your userid at the login prompt, then press ENTER. Your userid is case-sensitive, so be sure you type it exactly as your system administrator instructed.
3. Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.
4. If you provided correct userid and password then you would be allowed to enter into the system. Read the informatand messages that come up on the screen something as below.

   login : amrood
   amrood's password:
   Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73

You would be provided with a command prompt (sometime called $ prompt) where you would type your all the commands. For example to check calendar you need to type **cal** command as follows:

```
$ cal
           June 2009
    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
$
```

Change Password:

All Unix systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from hackers and crackers. Here are the steps to change your password:

1.To start, type **passwd** at command prompt as shown below.
2.Enter your old password the one you're currently using.
3. Type in your new password. Always keep your password complex enough so that no body can guess it. But make sure, you remember it.
4. You would need to verify the password by typing it again.

$ passwd
Changing password for amrood
(current) Unix password:******
New UNIX password:*******
Retype new UNIX password:*******
passwd: all authentication tokens updated successfully
$

Listing Directories and Files:

All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

You can use **ls** command to list out all the files or directories available in a directory. Following is the example of using **ls** command with **-l** option.

$ ls -l
total 19621
drwxrwxr-x 2 amrood amrood 4096 Dec 25 09:59 uml
-rw-rw-r— 1 amrood amrood 5341 Dec 25 08:38 uml.jpg
drwxr-xr-x 2 amrood amrood 4096 Feb 15 2006 univ
drwxr-xr-x 2 root root 4096 Dec 9 2007 urlspedia
-rw-r--r-- 1 root root 276480 Dec 9 2007 urlspedia.tar
drwxr-xr-x 8 root root 4096 Nov 25 2007 usr
-rwxr-xr-x 1 root root 3192 Nov 25 2007 webthumb.php
-rw-rw-r-- 1 amrood amrood 20480 Nov 25 2007 webthumb.tar

Who Are You

While you're logged in to the system, you might be willing to know : **Who am I**?
The easiest way to find out "who you are" is to enter the **whoami** command:

$ whoami
amrood
$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu ttyp2 Oct 4 09:08 (calliope)
qadir ttyp4 Oct 8 12:09 (dent)

Logging Out:

When you finish your session, you need to log out of the system to ensure that nobody else accesses your files while masquerading as you.

To log out:

Just type **logout** command at command prompt, and the system will clean up everything and break the connection

Listing Files:

To list the files and directories stored in the current directory. Use the following command:

$ls

```
bin  hosts  lib  res.03
ch07  hw1 pub test_results
ch07 .bak  hw2 res.01 users
docs  hw3  res.02 work
```

The command **ls** supports the **-1** option which would help you to get more information about the listed files:

$ls -l

```
total 1962188
drwxrwxr-x 2  amrood amrood 4096 Dec 25 09:59 uml
-rw-rw-r--    1 amrood amrood  5341 Dec 25 08:38 uml.jpg
drwxr-xr-x   2 amrood amrood 4096 Feb 15 2006 univ
drwxr-xr-x   2 root root 4096 Dec 9 2007 urlspedia
-rw-r--r--     1 root root 276480 Dec 9 2007 urlspedia.tar
drwxr-xr-x   8 root root 4096 Nov 25 2007 usr
drwxr-xr-x   2 200 300 4096 Nov 25 2007 webthumb-1.01
-rwxr-xr-x    1 root root 3192 Nov 25 2007 webthumb.php
-rw-rw-r--    1 amrood amrood 20480 Nov 25 2007 webthumb.tar
-rw-rw-r--    1 amrood amrood 5654 Aug 9 2007 yourfile.mid
-rw-rw-r--    1 amrood amrood 166255 Aug 9 2007 yourfile.swf
drwxr-xr-x  11 amrood amrood 4096 May 29 2007 zlib-1.2.3
```

Hidden Files:

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files:

- **.profile:** the Bourne shell (sh) initialization script
- **.kshrc:** the Korn shell (ksh) initialization script
- **.cshrc:** the C shell (csh) initialization script
- **.rhosts:** the remote shell configuration file

To list invisible files, specify the -a option to ls:

$ ls -a

```
  .  .profile  docs   lib  test_ results
  .. .rhosts    hosts pub users
     .emacs    bin    hw1 res.01 work
     .exrc      ch07  hw2 res.02
     .kshrc     ch07  .bak   hw3 res.03
```

Creating Files:

You can use **vi** editor to create ordinary files on any Unix system. You simply need to give following command:

$ vi filename

Above command would open a file with the given filename. You would need to press key **i** to come into edit mode. Once you are in edit mode you can start writing your content in the file as below:

This is unix file....I created it for the first time.....
I'm going to save this content in this file.

Once you are done, do the following steps:

- Press key esc to come out of edit mode.
- Press two keys Shift + ZZ together to come out of the file completely

Now you would have a file created with filemame in the current directory

$ vi filename

Editing Files:

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows:

$ vi filename

Once file is opened, you can come in edit mode by pressing key i and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key esc and then you can use following keys to move inside a file:

- l key to move to the right side.
- h key to move to the left side.
- k key to move up side in the file.
- j key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use i key to come in edit mode. Edit the file, once you are done press esc and finally two keys Shift + ZZ together to come out of the file completely.

Display Content of a File:

You can use **cat** command to see the content of a file. Following is the simple example to see the content of above created file:

$ cat filename

This is unix file....I created it for the first time.....
I'm going to save this content in this file.

Counting Words in a File:

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is the simple example to see the information about above created file:

$ wc filename
2 19 103 filename

Here is the detail of all the four columns:

1. First Column: represents total number of lines in the file.
2. Second Column: represents total number of words in the file.
3. Third Column: represents total number of bytes in the file. This is actual size of the file

4. Fourth Column: represents file name

You can give multiple files at a time to get the information about those file. Here is simple syntax:

$ wc filename1 filename2 filename3

Copying Files:

To make a copy of a file use the cp command. The basic syntax of the command is:

$ cp source_file destination_file

Following is the example to create a copy of existing file **filename**.

$ cp filename copyfile

Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

Renaming Files:

To change the name of a file use the **mv** command. Its basic syntax is:

$ mv old_file new_file

Following is the example which would rename existing file **filename** to **newfile**:

$ mv filename newfile

The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory

Deleting Files:

To delete an existing file use the **rm** command. Its basic syntax is:

$ rm filename

rm command.

Following is the example which would completely remove existing file **filename**:

$ rm filename

You can remove multiple files at a tile as follows:

$ rm filename1 filename2 filename3

Unix Directories:

A directory is a file whose sole job is to store file names and related information. All files whether ordinary, special, or directory, are contained in directories.

UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character (/), and all other directories are contained below it.

Home Directory:

The directory in which you find yourself when you first login is called your home directory.

You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command:

$cd ~

Here **~** indicates home directory. If you want to go in any other user's home directory then use the following command:

$cd ~username

To go in your last directory you can use following command:

$cd -

Absolute/Relative Pathnames:

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /.

These are some example of absolute filenames.
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/Os3
A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user amrood' home directory, some pathnames might look like this:
chem/notes
personal/res
To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory:
$pwd
/user0/home/amrood

Listing Directories:

To list the files in a directory you can use the following syntax:

$ls dirname
Following is the example to list all the files contained in /usr/local directory:
$ls /usr/local

X11	bin	imp	jikes	sbin
ace	doc	include	lib	share
atalk	etc	info	man	ami

Creating Directories:

Directories are created by the following command:
$mkdir dirname
Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command:
$mkdir mydir
Creates the directory mydir in the current directory. Here is another example:
$ mkdir /tmp/test-dir
This command creates the directory test-dir in the /tmp directory. The **mkdir** command produces no output if it successfully creates the requested directory.
If you give more than one directory on the command line, mkdir creates each of the directories. For example:
$ mkdir docs pub
Creates the directories docs and pub under the current directory.

Creating Parent Directories:

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, mkdir issues an error message as follows:
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test";
No such file or directory
In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example:
$mkdir -p /tmp/amrood/test
Above command creates all the required parent directories.
Removing Directories:
Directories can be deleted using the **rmdir** command as follows:

$rmdir dirname
You can create multiple directories at a time as follows:
$rmdir dirname1 dirname2 dirname3
Above command removes the directories dirname1, dirname2, and dirname2 if they are empty. The rmdir command produces no output if it is successful.

Changing Directories:

You can use the **cd** command to do more than change to a home directory: You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as follows:
$cd dirname
Here, dirname is the name of the directory that you want to change to. For example, the command:
$cd /usr/local/bin
Changes to the directory /usr/local/bin. From this directory you can cd to the directory /usr/home/amrood using the following relative path:
$cd ../../home/amrood

Renaming Directories:

The mv (move) command can also be used to rename a directory. The syntax is as follows:
$mv olddir newdir
You can rename a directory **mydir** to **yourdir** as follows:
$mv mydir yourdir

Unix File Permission:

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

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

The Permission Indicators:

While using **ls -l** command it displays various information related to file permission as follows:
$ls -l /home/amrood
-rwxr-xr-- 1 amrood users 1024 Nov 2 00:10 myfile
drwxr-xr--- 1 amrood users 1024 Nov 2 00:10 mydir
Here first column represents different access mode ie. permission associated with a file or directory.
The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x):

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

- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

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

1. Read:
Grants the capability to read ie. view the contents of the file.
2. Write:
Grants the capability to modify, or remove the content of the file.
3. Execute:
User with execute permissions can run a file as a program.
Directory Access Modes:
Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:
1. Read:
Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.
2. Write:
Access means that the user can add or delete files to the contents of the directory.
3. Execute:
Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.
A user must have execute access to the **bin** directory in order to execute ls or cd command.
Changing Permissions:
To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.
Using chmod in Symbolic Mode:
The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

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

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:
$ls -l testfile
-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile
Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx 1 amrood users 1024 Nov 2 00:10 testfile
$chmod g=r-x testfile
$ls -l testfile
-rw-r-xrwx 1 amrood users 1024 Nov 2 00:10 testfile

Here's how you could combine these commands on a single line:
$chmod o+wx,u-x,g=r-x testfile
$ls -l testfile
-rw-r-xrwx 1 amrood users 1024 Nov 2 00:10 testfile

Using chmod with Absolute Permissions:

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.
Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

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

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:
$ls -l testfile
-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile
Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x 1 amrood users 1024 Nov 2 00:10 testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx 1 amrood users 1024 Nov 2 00:10 testfile
$chmod 043 testfile
$ls -l testfile
----r---wx 1 amrood users 1024 Nov 2 00:10 testfile

Changing Owners and Groups:

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.
Two commands are available to change the owner and the group of files:
1. **chown:** The chown command stands for "change owner" and is used to change the owner of a file.
2. **chgrp:** The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership:

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

$ chown user filelist

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

Following example:

$ chown amrood testfile

Changing Group Ownership:

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

$ chgrp group filelist

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

Following example:

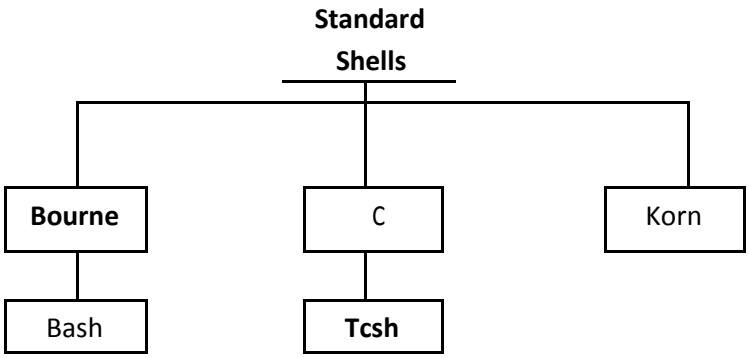
$ chgrp special testfile

UNIT-II

UNIT-II
SHELLS

The shell is the part of the UNIX that is most visible to the user. It receives and interprets the commands entered by the user. In many respects, this makes it the most important component of the UNIX structure.

To do anything in the system, we should give the shell a command. If the command requires a utility, the shell requests that the kernel execute the utility. If the command requires an application program, the shell requests that it be run. The standard shells are of different types as shown below:



There are two major parts to a shell. The first is the interpreter. The interpreter reads your commands and works with the kernel to execute them. The second part of the shell is a programming capability that allows you to write a shell (command) script.

A **shell script** is a file that contains shell commands that perform a useful function. It is also known as shell program.

Three additional shells are used in UNIX today. The Bourne shell, developed by Steve Bourne at the AT&T labs, is the oldest. Because it is the oldest and most primitive, it is not used on many systems today. An enhanced version of Bourne shell, called Bash (Bourne again shell), is used in Linux.

The C shell, developed in Berkeley by Bill Joy, received its name from the fact that its commands were supposed to look like C statements. A compatible version of C shell, called **tcsh** is used in Linux.

The Korn shell, developed by David Korn also of the AT&T labs, is the newest and most powerful. Because it was developed at AT&T labs, it is compatible with the Borne shell.

NIX SESSION:

A UNIX session consists of logging in to the system and then executing commands to accomplish our work. When our work is done, we log out of the system. This work flow is shown in the following flowchart:

When you log in you are in one of the five shells. The system administrator determines which shell you start in by an entry in the password file (/etc/passwd). Even though your start up shell is determined by the system administrator, you can always switch to another shell. The following example shows how to move to other shells:

```
$ bash      # Move to Bash shell
$ ksh       # Move to Korn shell
$ csh       # Move to C shell
```

LOGIN SHELL VERIFICATION:

UNIX contains a system variable, SHELL that identifies the path to your login shell. You can check it with the command as follows:

```
$ echo $SHELL
/bin/ksh
```

Note: the variable name is all uppercase.

CURRENT SHELL VERIFICATION:

Your current shell may or may not be your login shell. To determine what your current shell is, you can use the following command. Note, however that this command works only with the Korn and Bash shells; it does not work with the C shell.

SHELL RELATIONSHIPS:

When you move from one shell to another, UNIX remembers the path you followed by creating a parent-child relationship. Your login shell is always the most senior shell in the relationship – the parent or grandparent depending on how many shells you have used.

Let us assume that your login shell is Korn shell. If you then move to the Bash shell, the Korn shell is the parent and Bash shell is the child. If later in the session you move to the C shell, the C shell is the child of Bash shell and the Bash shell is the child of Korn shell.

To move from child shell to a parent shell we use the **exit** command. When we move up to parent shell, the child shell is destroyed – it no longer exists. Should you create a child, an entirely new shell is created.

LOGOUT:

To quit the session – that is, to log out of the system – you must be at the original login shell. You cannot log out from a child. If you try to log out from a child, you will get an error message. The Korn shell and Bash shell both display a not-found message such as “logout not found”. The C shell is more specific: it reports that you are not in login shell.

The correct command to end the session at the login shell is **logout**, but the **exit** command also terminates the session

STANDARD STREAMS:

UNIX defines three standard streams that are used by commands. Each command takes its input from a stream known as **standard input**. Commands that create output send it to a stream known as **standard output**. If an executing command encounters an error, the error message is sent to **standard error**. The standard streams are referenced by assigning a descriptor to each stream. The descriptor for standard input is 0 (zero), for standard input is 1, and for standard output is 2.

There is a default physical file associated with each stream: standard input is associated with the keyboard, standard output is associated with monitor and standard error is also associated with the monitor. We can change the default file association using pipes or redirection.

UNIX defines three standard streams that are used by commands. Each command takes its input from a stream known as **standard input**. Commands that create output send it to a stream known as **standard output**. If an executing command encounters an error, the error message is sent to **standard error**. The standard streams are referenced by assigning a descriptor to each stream. The descriptor for standard input is 0 (zero), for standard input is 1, and for standard output is 2.

There is a default physical file associated with each stream: standard input is associated with the keyboard, standard output is associated with monitor and standard error is also associated with the monitor. We can change the default file association using pipes or redirection.

REDIRECTION:

It is the process by which we specify that a file is to be used in place of one of the standard files. With input files, we call it input redirection; with output files, we call it as output redirection; and with error file, we call it as error redirection.

Redirecting Input: we can redirect the standard input from the keyboard to any text file. The input redirection operator is the less than character (<). Think of it as an arrow pointing to a command, meaning that the command is to get its input from the designated file. There are two ways to redirect the input as shown below:

command 0< file1 or ***command < file1***

The first method explicitly specifies that the redirection is applied to standard input by coding the 0 descriptor. The second method omits the descriptor. Because there is only-one standard input, we can omit it. Also note that there is no space between the descriptor and the redirection symbol.

Redirecting Output:

When we redirect standard output, the commands output is copied to a file rather than displayed on the monitor. The concept of redirected output appears as below:

command 1>	file1	or	command > file1
command 1>|	file1	or	command >| file1
command 1>>	file1	or	command >> file1

There are two basic redirection operators for standard output. Both start with the greater than character (>). Think of the greater than character as an arrow pointing away from the command and to the file that is to receive the output.

Which of the operators you use depends on how you want to output the file handled. If you want the file to contain only the output from this execution of the command, you use one greater than token (>). In this case when you redirect the output to a file that does not exist, UNIX creates it and writes the output.

If the file already exists the action depends on the setting of a UNIX option known as **noclobber**. When the noclobber option is turned on, it prevents redirected output from destroying an existing file. In this case you get an error message which is as given in below example.

$ **who > whoOct2**
ksh: whoOct2: file already exists

If you want to override the option and replace the current file's contents with new output, you must use the redirection override operator, greater than bar (>|). In this case, UNIX first empties the file and then writes the new output to the file. The redirection override output is as shown in the below example:

$ **who > whoOct2**
$ **more whoOct2**
abu52408 ttyq3 Oct 2 15:24 (atc2west-171.atc.fhda.edu)

On the other hand if you want to append the output to the file, the redirection token is two greater than characters (>>). Think of the first greater than as saying you want to redirect the output and the second one as saying that you want to go to the end of the file before you start outputting.

When you append output, if the file doesn't exist, UNIX creates it and writes the output. If it is already exists, however, UNIX moves to the end of the file before writing any new output.

Redirecting errors:

One of the difficulties with the standard error stream is that it is, by default, combined with the standard output stream on the monitor. In the following example we use the long list (ls) command to display the permissions of two files. If both are valid, one displays after the other. If only one is valid, it is displayed but ***ls*** display an error message for the other one on the same monitor.

$ **ls –l file1 noFile**
Cannot access noFile: No such file or directory
-rw-r- - r- - 1 gilberg staff 1234 Oct 2 18:16 file1

We can redirect the standard output to a file and leave the standard error file assigned to the monitor.

REDIRECTING TO DIFFERENT FILES:

To redirect to different files, we must use the stream descriptors. Actually when we use only one greater than sign, the system assumes that we are redirecting the output (descriptor 1). To redirect them both, therefore, we specify the descriptor (0, 1, or 2) and then the redirection operator as shown in below example:

$ **ls –l file noFile 1> myStdOut 2> myStdErr**

$ **more myStdOut**

-rw-r- - r- - 1 gilberg staff 1234 Oct 2 18:16 file1

$ **more myStdErr**

Cannot open noFile: No such file or directory

The descriptor and the redirection operator must be written as consecutive characters; there can be no space between them. It makes no difference which one you specify first.

REDIRECTING TO ONE FILE:

If we want both outputs to be written to the same file, we cannot simply specify the file name twice. If we do, the command fails because the file is already open. This is the case as given in the following example:

$ **ls –l file1 noFile 1> myStdOut 2> myStdOut**

ksh: myStdOut: file already exists

If we use redirection override operator, the output file contains only the results of the last command output which is as given below:

$ **ls –l file1 noFile 1>| myStdOut 2>| myStdOut**

$ **ls myStdOut**

Cannot open file noFile: No such file or directory

To write all output to the same file, we must tell UNIX that the second file is really the same as the first. We do this with another operator called **and** operator (&). An example of the substitution operator is shown as follows:

$ **ls –l file1 noFile 1> myStdOut 2>& 1**

$ **more myStdOut**

Cannot open file noFile: No such file or directory

-rw-r- - r- - 1 gilberg staff 1234 Oct 2 18:16 file1

The following table shows the redirection differences between the shells:

Type	Korn and Bash Shells	C Shell
Input	*0< file1* or *< file1*	< file1
Output	*1> file1* or *> file1* *1>| file1* or *>| file1* *1>> file1* or *>> file1*	*> file1* >| file1 >> file1
Error	*2> file2* *2>| file2* *2>> file2*	Not Supported Not Supported Not Supported
Output & Error (different files)	*1> file1* 2> *file2* *> file1* 2> *file2*	Not Supported Not Supported
Output & Error (same files)	*1> file1* 2> *& 1* *> file1* 2> *& 1* *1>| file1* 2> *& 1*	>& file1 >& file1 >&! file1

PIPES:

We often need to uses a series of commands to complete a task. For example if we need to see a list of users logged in to the system, we use the **who** command. However if we need a hard copy of the list, we need two commands. First we use **who** command to get the list and store the result in a file using redirection. We then use the **lpr** command to print the file. This sequence of commands is shown as follows:

who > file1

lpr file1

We can avoid the creation of the intermediate file by using a pipe. Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command. The first command must be able to send its output to standard output; the second command must be able to read its input from standard input. This command sequence is given as follows:

$ **who | lpr**

Think of the pipe as a combination of a monitor and a keyboard. The input to the pipe operator must come from standard output. This means that the command on the left that sends output to the pipe must write its output to standard output.

A pipe operator receives its input from standard output and sends it to the next command through standard input. This means that the left command must be able to send data to standard output and the right command must be able to receive data from standard input.

The token for a pipe is the vertical bar (|). There is no standard location on the keyboard for the bar. Usually you will find it somewhere on the right side, often above the return key.

Pipe is an operator not a command. It tells the shell to immediately take the output of the first command, which must be sent to the standard output, and turn it into input for the second command, which must get its input from standard input.

TEE COMMAND:

The **tee** command copies standard input to a standard output and at the same time copies it to one or more files. The first copy goes to standard output, which is usually the monitor. At the same time, the output is sent to the optional files specified in the argument list.

The format of tee command is **tee options file-list**

The tee command creates the output files if they do not exist and overwrites them if they already exist. To prevent the files from being overwritten, we can use the option –a, which tells tee to append the output to existing files rather than deleting their current content.

Note however, that the append option does not apply to the standard output because standard output is always automatically appended. To verify the output to the file, we use more to copy it to the screen.

COMMAND EXECUTION:

Nothing happens in a UNIX shell until a command executed. When a user enters a command on the command line, the interpreter analyzes the command and directs its execution to a utility or other program.

Some commands are short and can be entered as a single line at the command prompt. We have seen several simple commands such as cal and date, already. At other times, we need to combine several commands.

There are four syntactical formats for combining commands in to one line: sequenced, grouped, chained, and conditional.

SEQUENCED COMMANDS:

A sequence of commands can be entered on one line. Each command must be separated from its predecessor by a semicolon. There is no direct relationship between the commands; that is one command does not communicate with the other. They simply combined in to one line and executed.

Example of a command sequence assumes that we want to create a calendar with a descriptive title. This is easily done with a sequence as shown below:

$ **echo "\n Goblins & Ghosts\n Month" > Oct2000; cal 10 2000 >> Oct2000**

GROUPED COMMANDS:

We redirected the output of two commands to the same file. This technique gives us the intended results, but we can do it more easily by grouping the commands. When we group commands, we apply the same operation to the group. Commands are grouped by placing them in parentheses.

CHAINED COMMANDS:

In the previous two methods of combining commands into one line, there was no relationship between the commands. Each command operated independently of the other and only shared a need to have their output in one file. The third method of combining commands is to pipe them. In this case however, there is a direct relationship between the commands. The output of the first becomes the input of the second.

CONDITIONAL COMMANDS:

We can combine two or more commands using conditional relationships. There are two shell logical operators, ***and*** (***&&***) and ***or*** (**||**). In general when two commands are combined with a logical ***and***, the second executes only if the first command is successful.

Conversely if two commands are combined using the logical ***or***, the second command executes only if the first fails.

Example:

$ **cp file1 tempfile && echo " Copy successful"**
Copy successful
$ **cp noFile tempfile || echo "Copy failed"**
Copy failed

COMMAND SUBSTITUTION:

When a shell executes a command, the output is directed to standard output. Most of the time standard output is associated with the monitor. There are times, however such as when we write complex commands or scripts, which we need to change the output to a string that we can store in another string or a variable.

Command substitution provides the capability to convert the result of a command to a string. The command substitution operator that converts the output of a command to a string is a dollar sign and a set of parentheses (Figure 2.1).

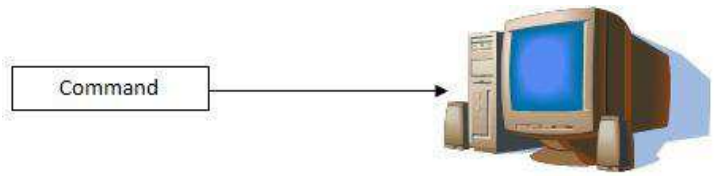


FIGURE 2.1(a): WITHOUT COMMAND SUBSTITUTION

Open parentheses Close parentheses

$(command) String

FIGURE 2.1(b): WITH COMMAND SUBSTITUTION

As shown in the figure 2.1, to invoke command substitution, we enclose the command in a set of parentheses preceded by a dollar sign ($). When we use command substitution, the command is executed, and its output is created and then converted to a string of characters.

COMMAND LINE EDITING:

The phrase “to err is human” applies with a harsh reality to all forms of computing. As we enter commands in UNIX command line, it is very easy to err. As long as we haven’t keyed Return and we notice the mistake, we can correct it. But what if we have keyed Return?

There are two ways we can edit previous commands in the Korn and Bash shells and one way in C shell.

In the Korn and Bash shells we can use the history file or we can use command-line editing. The history file is a special UNIX file that contains a list of commands use during a session. In the C shell, we can use only the history file. The following table summarizes the command line editing options available.

Method	Korn Shell	Bash Shell	C Shell
Command line	√	√	
History File	√	√	√

COMMAND LINE EDITING CONCEPT:

As each command is entered on the command line, the Korn shell copies it to a special file. With command line editing, we can edit the commands using either vi or emacs without opening the file. It's as though the shell keeps the file in a buffer that provides instant access to our commands. Whenever a command completes, the shell moves to the next command line and waits for the next command.

EDITOR SELECTION:

The system administrator may set the default command line editor, most likely in ***/etc/profile***. You can switch it, however by setting it yourself. If you set it at command line, it is set for only the current session. If you add it to your login file, it will be changed every time you log in. During the session you can also change it from one to another and back whenever you like.

To set the editor, we use the ***set*** command with the editor as shown in the next example; you would use only one of the two commands.

$ **set –o vi** # Turn on vi editor
$ **set –o emacs** # Turn on emacs editor

VI COMMAND LINE EDITOR:

We cannot tell which editor we are using at the command line simply by examining the prompt. Both editors return to the shell command line and wait for response. If quickly becomes obvious however, by the behavior of the command line.

The vi command line editor opens in the insert mode. This allows us to enter commands easily. When we key Return, the command is executed and we return to the vi insert mode waiting for input.

VI EDIT MODE

Remember that the vi editor treats the history file as though it is always open and available. Because vi starts in the insert mode, however to move to the vi command mode we must use the Escape key. Once in the vi command mode, we can use several of the standard vi commands. The most obvious commands that are not available are the read, write and quit commands.

The basic commands that are available are listed in the table below:

Category	Command	Description
Adding Text	I	Inserts text before the current character.
	I	Inserts text at the beginning of the current line.
	A	Appends text after the current character.
	A	Appends text at the end of the current line.
Deleting Text	X	Deletes the current character.
	Dd	Deletes the command line.
Moving Cursor	H	Moves the cursor one character to the left.
	L	Moves the cursor one character to the right.
	0	Moves the cursor to the beginning of the current line.
	$	Moves the cursor to the end of the current line.
	K	Moves the cursor one line up.
	J	Moves the cursor one line down.
	-	Moves the cursor to the beginning of the previous line.
	+	Moves the cursor to the beginning of the next line.
Undo	U	Undoes only the last edit.
	U	Undoes all changes on the current line.
Mode	<esc>	Enters command mode.
	i, I, a, A	Enters insert mode.

Only one line is displayed at any time. Any changes made to the line do not change the previous line in the file. However, when the edited line is executed, it is appended to the end to the end of the file.

USING THE COMMAND LINE EDITOR:

There are several techniques for using the command line editor.

Execute a Previous Line: To execute a previous line, we must move up the history file until we find the line. This requires the following steps:

1. Move to command mode by keying Escape (esc).
2. Move up the list using the Move-up key (k)
3. When the command has been located, key Return to execute it.

After the command has been executed, we are back at the bottom of the history file in the insert mode.

Edit and Execute a Previous Command: Assume that we have just executed the **more** command with a misspelled file name. In the next example, we left off the last character of the filename, such as “file” rather than “file1”.

1. Move to the command mode by keying Escape (esc).
2. Using the Move-up key (k), recall the previous line.
3. Use the append-at-end command (A) to move the cursor to the end of the line in the insert mode.
4. Key the missing character and Return.

After executing the line, we are in the insert mode.

JOB CONTROL:

One of the important features of a shell is job control.

Jobs:

In general a job is a user task run on the computer. Editing, sorting and reading mail are all examples of jobs. However UNIX has a specific definition of a job. *A job is a command or set of commands entered on one command line*. For example:

$ **ls**
$ **ls | lpr**

Both are jobs.

Foreground and Background Jobs:

Because UNIX is a multitasking operating system, we can run more than one job at a time. However we start a job in the foreground, the standard input and output are locked. They are available exclusively to the current job until it completes. This means only one job that needs these files can run at a time. To allow multiple jobs, therefore, UNIX defines two types of jobs: foreground and background.

FOREGROUND JOBS:

A foreground job is any job run under the active supervision of the user. It is started by the user and may interact with the user through standard input and output. While it is running, no other jobs may be started. To start a foreground job, we simply enter a command and key Return. Keying Return at the end of the command starts it in the foreground.

Suspending a foreground job *While a foreground job is running it can be suspended. For example, while you are running a long sort in the foreground, you get a notice that you have mail.*

To read and respond to your mail, you must suspend the job. After you are through the mail you can then restart the sort. To suspend the foreground job, key ctrl+z. To resume it, use the foreground command (**fg**).

Terminating a foreground job If for any reason we want to terminate (kill) a running foreground job, we use the cancel meta-character, *ctrl+c*. After the job is terminated, we key Return to activate the command line prompt. If the job has been suspended, it must first be resumed using the foreground command.

BACKGROUND JOBS:

When we know a job will take a long time, we may want to run it in the background. Jobs run in the background free the keyboard and monitor so that we may use them for other tasks like editing files and sending mail.

Note: Foreground and Background jobs share the keyboard and monitor.

Any messages send to the monitor by the background job will therefore be mingled with the messages from foreground job.

Suspending, Restarting and Terminating Background jobs To suspend the background job, we use the **stop** command. To restart it, we use the **bg** command. To terminate the background job, we use the **kill** command. All three commands require the job number, prefaced with a percent sign (%).

Example:

$ **longjob.scr&**
[1] 1795841
$ **stop %1**
[1] + 1795841 stopped (SIGSTOP) longjob.scr&
$ **bg %1**
[1] longjob.scr&
$ **kill %1**
[1] + Terminated longjob.scr&

Moving between Background and Foreground To move a job between the foreground and background, the job must be suspended. Once the job is suspended, we can move it from the suspended state to the background with the **bg** command. Because job is in the foreground, no job number is required. To move a background job to a foreground job, we use the **fg** command.

MULTIPLE BACKGROUND JOBS:

When multiple background jobs are running in the background, the job number is required on commands to identify which job we want to affect.

Jobs command

To list the current jobs and their status, we use the **jobs** command. This command lists all jobs. Whether or not they are running or stopped. For each job, it shows the job number, currency, and status, running or stopped.

JOB STATES:

At any time the job may be in one of the three states: foreground, background or stopped. When a job starts, it runs the foreground. While it is running, the user can stop it, terminate it, or let it run to completion. The user can restart a stopped job by moving it to either the foreground or background state. The user can also terminate a job. A terminated job no longer exists. To be terminated, a job must be running.

While job is running it may complete or exit. A job that completes has successfully finished its assigned tasks. A job that exits has determined that it cannot complete its assigned tasks but also cannot continue running because some internal status has made completion impossible. When a job terminates either because it is done or it must exit, it sets a status code that can be checked by the user. The following figure 2.2 summarizes the job states:

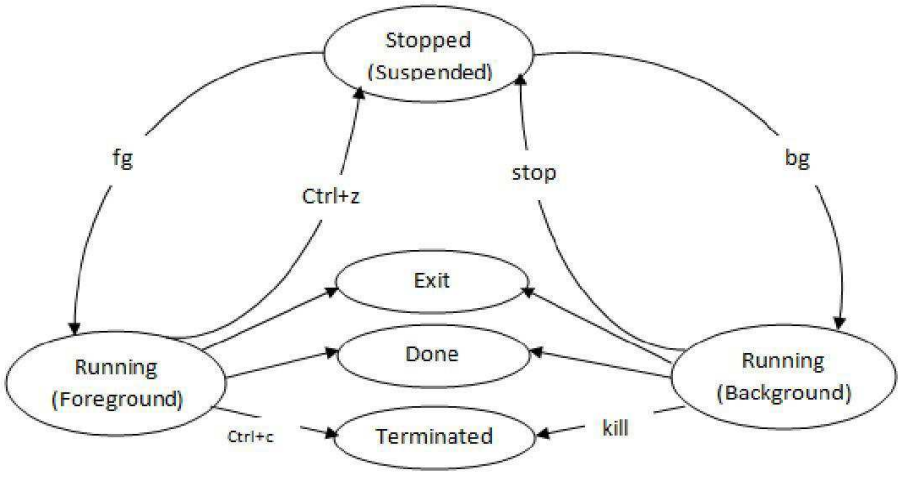


FIGURE 2.2 JOB STATES

Process ID

Job numbers are related to user session and the terminal; they are not global. UNIX assigns another identification, which is global in scope, to jobs or processes. It is called the process identifier, or PID. The **ps** command displays the current PIDs associated with the terminal which it is entered.

ALIASES:

An alias provides means of creating customized commands by assigning a name to a command. Aliases are handled differently in each shell.

Aliases in Korn and Bash Shells:

In the Korn and Bash shells, an alias is created by using the **alias** command. Its format is ***alias name=command-definition*** where alias is the command keyword, name is the alias being created, and command-definition is the code.

Example:

Renaming Commands One of the common uses of aliases is to create a more intuitive name for a command.

$ **alias dir=ls**
$ **dir**
TheRaven file1 longJob.scr
TheRaven1 fileOut loop.scr

Alias of command with Options An even better example is the definition of a command with options. We define a directory command with the long list option.

Example:

$ **alias dir=`ls -l`**

$ **dir**

Total 6

-rw- - - - - - - 1 gilberg staff 5782 10 16:19 TheRaven

Alias of Multiple Command Lines Often a process requires more than one command. As long as the commands are considered one line in UNIX, they can be assigned as alias. Because some list output can be larger one screen of output, it is good idea to pipe the output to **more**.

$ **alias dir="ls –l | more"**

$ **dir**

Total 6

-rw- - - - - - - 1 gilberg staff 5782 10 16:19 TheRaven

. . . .

-rw-r- - r - - 1 gilberg staff 149 Apr 18 2000 loop.scr

Using an Alias in and Alias Definition It is possible to create an alias definition using a definition. There is one danger in this usage; however, if a definition uses multiple aliases and one of them refers to another one, the definition may become recursive and it will bring down the shell. For this reason, it is recommended that only one definition can be used in defining a new alias definition.

Example:

$ **alias dir=ls**

$ **alias lndir=`dir –l | more`** $ **lndir**

Total 6

-rw- - - - - - - 1 gilberg staff 5782 10 16:19 TheRaven

. . . .

-rw-r- - r - - 1 gilberg staff 149 Apr 18 2000 loop.scr

ARGUMENTS TO ALIAS COMMANDS:

An argument may be passed to an alias as long as it is not ambiguous. Arguments are added after the command. The arguments may contain wildcards as appropriate. For example, we can pass a file to the list command so that it lists only the file(s) requested.

$ **alias fl="ls –l"**
$ **fl f***

```
+ ls –l fgLoop.scr      file1    fileOut
-rwx- - - - -    1    gilberg        staff    175    13    10:38    fgLoop.scr
-rw-r - -r- - -    1    gilberg        staff    15    17    2000    file1
-rw-r - -r- - -    1    gilberg        staff    395    9    20:00    fileOut
```

Expanded commands are displayed starting with a plus followed by the command with its arguments filled in. Sometimes arguments can be ambiguous. This usually happens when multiple commands are included in one alias.

Listing aliases:

The Korn and Bash shells provide a method to list all aliases and to list a specific alias. Both use the alias command. To list all aliases, we use the alias command with no arguments. To list a specific command, we use the alias command with one argument, the name of the alias command. These variations are shown below:

```
$ alias
autoload = 'typeset –fu'
cat = /sbin/cat
dir = 'echo '\' 'listing for gilberg'\' '; ls –l | more'
fl = 'ls –l | more'
. . .
Stop = 'kill –STOP'
suspend = 'kill –STOP $$'
$ alias dir
dir = 'echo '\' 'listing for gilberg'\' '; ls –l | more'
```

Removing Aliases:

Aliases are removed by using the **unalias** command. It has one argument, a list of aliases to be removed. When it is used with the all option (-a), it deletes all aliases. You should be very careful, however with this option: It deletes all aliases, even those defined by the system administrator. For this reason, some system administrators disable this option. The following example demonstrates only the remove name argument.

```
$ alias dir
dir = 'echo '\' 'listing for gilberg'\' '; ls –l | more'

$ unalias dir
$ alias dir
dir: alias not found
```

Aliases in the C Shell:

C shell aliases differ from Korn shell aliases in format but not in function. They also have more powerful set of features, especially for argument definition. The syntax for defining a C shell alias differs slightly in that there is no assignment operator. The basic format is:

alias name definition

Example:

% **alias dir "echo Gilbergs Directory List; ls –l | more"**
% **dir**
Gilbergs Directory List
total 30
-rw- - - - - - - 1 gilberg staff 5782 Sep 10 16:19 TheRaven
. . .
-rw- r - - r- - - 1 gilberg staff 149 Apr 18 2000 teeOut1
-rw- r - - r- - - 1 gilberg staff 149 Apr 18 2000 teeOut2

Arguments to Alias Command:

Unlike Korn shell arguments are positioned at the end of the generated command, the C shell allows us to control the positioning. The following table contains the position designators used for alias arguments.

Designator	Meaning
\!*	Position of the only argument.
\!^	Position of the first argument.
\!$	Position of the last argument.
\!:n	Position of the n^{th} argument.

When we wrote the file list alias in the Korn shell, the argument was positioned at the end, where it caused a problem.

Listing Aliases Just like the Korn shell, we can list a specific alias or all aliases. The syntax for the two shells is identical.

Example:

% **alias**
cpto cp \!:1 \!:$
dir echo Gilbergs Directory List; ls –l | more
f1 ls –l \!* | more

Removing Aliases:

The C shell uses the **unalias** command to remove one or all aliases. The following example shows the unalias command in C shell:

% **unalias f1**
% **alias**
cpto cp \!:1 \!:$
dir echo Gilbergs Directory List; ls –l | more

The following table summarizes the use of aliases in the three shells.

Feature	**Korn and Bash**	**C**
Define	$ alias x=command	% alias x command
Argument	Only at the end	Anywhere
List	$ alias	% alias
Remove	$ unalias x y z	% unalias x y z
Remove All	$ unalias –a	% unalias *

VARIABLES TYPES AND OPTIONS:

VARIABLES:

A variable is a location in memory where values can be stored. Each shell allows us to ***create***, ***store*** and ***access*** values in variables. Each shell variable must have a name. The name of a variable must start with an alphabetic or underscore (_) character. It then can be followed by zero or more alphanumeric or underscore characters.

There are two broad classifications of variables: (TYPES)

1. User-Defined
2. Predefined

User-Defined Variables:

User variables are not separately defined in UNIX. The first reference to a variable establishes it. The syntax for storing values in variables is the same for the Korn and Bash shells, but it is different for the C shell.

Predefined Variables:

Predefined variables are used to configure a user's shell environment. For example a system variable determines which editor is used to edit the command history. Other systems variables store information about the home directory.

Storing Data in Variables:

All three shells provide a means to store values in variables. Unfortunately, the C shell's method is different. The basic commands for each shell are presented in the table below:

Action	Korn and Bash	C Shell
Assignment	variable=value	set variable = value
Reference	$variable	%variable

Storing Data in the Korn and Bash Shells:

The Korn and Bash shells use the assignment operator, = , to store values in a variable. Much like an algebraic expression, the variable is coded first, on the left, followed by the assignment operator and then the value to be stored. There can be no spaces before and after the assignment operator; the variable, the operator, and the value must be coded in sequence immediately next to each other as in the following example:

varA=7

In the above example varA is the variable that receives the data and 7 is the value being stored in it. While the receiving field must always be a variable, the value may be a constant, the contents of another variable, or any expression that reduces to a single value. The following shows some examples of storing values in variables.

$ **x=23**
$ **echo $x**
23
$ **x=Hello**
$ **echo $x**
Hello
$ **x="Go Don's"**
$ **echo $x**
Go Don's

Storing Data in the C Shell:

To store the data in a variable in the C Shell, we use the **set** command. While there can be no spaces before and after the assignment operator in the Korn and Bash shells, C needs them. C also accepts the assignment without spaces before and after the assignment operator.

Example: %
set x = 23

% **echo $x**

23

% **set x = hello**

% **echo $x**

hello

Accessing a Variable:

To access the value of a variable, the name of the variable must be preceded by a dollar sign. We use the echo command to display the values. Example:

$ **x=23**

$ **echo The variable x contains $x**

The variable x contains

23 $ **x=hello**

$ **echo The variable x contains $x**

The variable x contains hello

Predefined Variables:

Predefined variables can be divided in to two categories: shell variable and environment variables. The shell variables are used to customize the shell itself. The environment variables control the user environment and can be exported to subshells. The following table lists the common predefined variables. In C shell, the shell variables are in lowercase letters, and the corresponding environmental variables are in uppercase letters.

Korn and Bash	**C**[a]	**Explanation**
CDPATH	cdpath	Contains the search path for **cd** command when the directory argument is a relative pathname.
EDIROT[b]	EDITOR	Path name of the command line editor
ENV		Pathname of the environment file
HOME[b]	home (HOME)	Pathname for the home directory
PATH[b]	path (PATH)	Search path for commands.
PS1	prompt	Primary prompt, such as $ and %
SHELL[b]	shell (SHELL)	Pathname of the login shell
TERM[b]	term (TERM)	Terminal type
TMOUT	autologout	Defines idle time, in seconds, before shell automatically logs you off.
VISUAL[b]	VISUAL	Pathname of the editor for command line editing. See EDITOR table entry.

[a]Shell variables are in lowercase; environmental variables are in uppercase

[b]Both a shell and an environmental variable

CDPATH:

The CDPATH variable contains a list of pathnames separated by colons (:) as shown in the example below:

:$HOME: /bin/usr/files

There are three paths in the preceding example. Because the path starts with a colon, the first directory is the current working directory. The second directory is our home directory. The third directory is an absolute pathname to a directory of files.

The contents of CDPATH are used by the **cd** command using the following rules:

1. If CDPATH is not defined, the **cd** command searches the working directory to locate the requested directory. If the requested directory is found, **cd** moves to it. If it is not found, **cd** displays an error message.
2. If CDPATH is defined as shown in the previous example, the actions listed below are taken when the following command is executed:

$ **cd reports**

a. The **cd** command searches the current directory for the *reports* directory. If it is found, the current directory is changed to *reports*.
b. If the reports directory is not found in the current directory, **cd** tries to find it in the home directory, which is the second entry in CDPATH. Note that the home directory may be the current directory. Again if the *reports* directory is found in the home directory, it becomes the current directory.
c. If the *reports* directory is not found in the home directory, **cd** tries to find in /bin/usr/files. This is the third entry in CDPATH. If the *reports* directory is found in /bin/usr/files, it becomes the current directory.
d. If the *reports* directory is not found in /bin/usr/files, **cd** displays an error message and terminates.

HOME:

The HOME variable contains the PATH to your home directory. The default is your login directory. Some commands use the value of this variable when they need the PATH to your home directory. For example, when you use the **cd** command without any argument, the command uses the value of the HOME variable as the argument. You can change its value, but we do not recommend you change if because if will affect all the commands and scripts that use it. The following example demonstrates how it can be changed to the current working directory. Note that because **pwd** is a command, it must be enclosed in back quotes.

$ **echo $HOME**

/mnt/diska/staff/gilberg

$ **oldHOME=$HOME**

$ **echo $oldHOME**

/mnt/diska/staff/gilberg

$ **HOME=$ (pwd)**

$ **echo $HOME**

/mnt/diska/staff/gilberg/unix13bash

$ **HOME=$oldHOME**

$ **echo $HOME**

/mnt/diska/staff/gilberg

PATH

The PATH variable is used to search for a command directory. The entries in the PATH variable must be separated by colons. PATH works just like CDPATH.

When the SHELL encounters a command, it uses the entries in the PATH variable to search for the command under each directory in the PATH variable. The major difference is that for security reasons, such as Trojan horse virus, we should have the current directory last.

If we were to set the PATH variable as shown in the below example, the shell would look for the **date** command by first searching the /bin directory, followed by the /usr/bin directory, and finally the current working directory.

$ **PATH=/bin: /usr/bin: :**

Primary Prompt (PSI Prompt)

The primary prompt is set in the variable *PS1* for the Korn and Bash shells and *prompt* for the C shell. The shell uses the primary prompt when it expects a command. The default is the dollar sign ($) for the Korn and Bash shells and the percent (%) sign for the C shell.

We can change the value of the prompt as in the example below:

$ **PS1=”KSH> “**

KSH> echo

$PS1 KSH>

KSH> **PS1=”$ “**

$

SHELL

The SHELL variable holds the path of your login shell.

TERM

The TERM variable holds the description for the terminal you are using. The value of this variable can be used by interactive commands such as **vi** or **emacs**. You can test the value of this variable or reset it.

Handling Variables:

We need to set, unset, and display the variables. Table below shows how this can be done for each shell.

Operation	**Korn and Bash**	**C Shell**
Set	var=value	set var = value (setenv var value)
Unset	unset var	unset var (unsetenv var)
Display One	echo $var	echo $var
Display All	set	set (setenv)

Korn and Bash Shells:

Setting and Unsetting: In the Korn and Bash shells, variables are set using the assignment operator as shown in the example:

$ **TERM=vt100**

To unset a variable, we use the **unset** command. The following example shows how we can unset the TERM variable.

$ **unset TERM**

Displaying variables: To display the value of an individual variable, we use the **echo** command:

$ **echo $TERM**

To display the variables that are currently set, we use the **set** command with no arguments:

$ **set**

C Shell

The C Shell uses the different syntax for changing its shell and environmental variables.

Setting and Unsetting: To set a shell variable it uses the **set** command; to set the environmental variable it uses the **setenv** command. These two commands demonstrated in the example below:

$ **set prompt = `CSH % `**

CSH % **setenv HOME /mnt/diska/staff/gilberg**

To unset the C shell variable, we use the **unset** command. To unset an environmental variable we use the **unsetenv** command.

Example:

CSH % **unset prompt**
unsetenv EDITOR

Displaying Variables: To display the value of the individual variable (both shell and environmental), we use the **echo** command. To list the variables that are currently set, we use the **set** command without an argument for the shell variables and the **setenv** command without an argument for the environmental variables. These commands are shown in the example below:

% **echo $variable-name** **# display one variable**

% **set** **# display all shell variables**

% **setenv** **# display all environmental variables**

OPTIONS:

The following table shows the common options used for the three shells.

Korn and Bash	C	Explanation
Noglob	noglob	Disables wildcard expansion.
Verbose	verbose	Prints commands before executing them.
Xtrace		Prints commands and arguments before executing them.
Emacs		Uses **emacs** for command-line editing.
Ignoreeof	ignoreeof	Disallows ctrl+d to exit the shell.
Noclobber	noclobber	Does not allow redirection to clobber existing file.
Vi		Users **vi** for command-line editing.

Global (noglob): The global option controls the expansion of wildcard tokens in a command. For example, when the global option is off, the list file (**ls**) command uses wildcards to match the files in a directory.

Thus the following command lists all files that start with 'file' followed by one character:

$ **ls file?**

On the other hand when the global option is on, wildcards become text characters and are not expanded. In this case only the file names 'file?' would be listed.

Print Commands (verbose and xtrace): There are two print options, verbose and xtrace that are used to print commands before they are executed. The verbose option prints the command before it is executed. The xtrace option expands the command arguments before it prints the command.

Command line Editor (emacs and **vi):** To specify that the **emacs** editor is to be used I the Korn shell, we turn on the emacs option. To specify that the **vi** editor is to be used in the Korn shell, we turn on the vi option. Note that these options are valid only in the Korn Shell.

Ignore end of file (ignoreeof): Normally, if end of file (ctrl+d) is entered at the command line, the shell terminates. To disable this action we can turn on the ignore end of file option, ignoreeof. With this option, end of file generates an error message rather than terminating the shell.

No Clobber Redirection (noclobber): when output or errors are directed to a file that already exists, the current file is deleted and replaced by a new file. To prevent this action we set the noclobber option.

Handling Options: To customize our shell environment we need to set, unset and display options; the following table shows the appropriate commands for each shell.

Operation	**Korn and Bash**	**C**
Set	set –o option	set option
Unset	set +o option	unset option
Display All	set -o	Set

Korn and Bash Shell Options:

Setting and Unsetting Options: To set and unset an option, we use the **set** command with –o and +o followed by the option identifier. Using the Korn shell format, we would set and unset the verbose option, as shown in the following example:

$ **set –o verbose** # Turn print commands option on
$ **set –o verbose** # Turn print commands option off

Display Options: To show all of the options (set or unset), we use the set command with an argument of –o. This option requests a list of all option names with their state, on or off.

```
$ set –o          # Korn Shell format: lists all options
```

C Shell Options:

Setting and Unsetting Options: In C shell, options are set with the **set** command and unset with the **unset** command, but without the minus sign in both cases. They are both shown in the following example:

```
$ set verbose     # Turn print commands option on
$ unset verbose   # Turn print commands option off
```

Displaying Options:

To display which options are set, we use the **set** command without an argument. However the C shell displays the setting of all variables including the options that are variables. The options are recognized because there is no value assigned to them: Only their names are listed. The next example shows the display options format:

```
$ set             # C shell format: lists all variables
```

SHELL / ENVIRONMENT CUSTOMIZATION:

UNIX allows us to customize the shells and the environment we use. When we customize the environment, we can extend it to include subshells and programs that we create.

There are four elements to customizing the shell and the environment. Depending on how we establish them, they can be temporary or permanent. Temporary customization lasts only for the current session. When a new session is started, the original settings are reestablished.

Temporary Customization:

It can be used to change the shell environment and configuration for the complete current session or for only part of a session. Normally we customize our environment for only a part of the session, such as when we are working on something special.

For example if we are writing a script it is handy to see the expanded commands as they are executed. We would do this by turning on the verbose option. When we are through writing the script, we would turn off the verbose option.

Any option changed during this session is automatically reset to its default when we log on the next time.

Permanent Customization:

It is achieved through the startup and shutdown files. Startup files are system files that are used to customize the environment when a shell begins. We can add customization commands and set customization variables by adding commands to the startup file. Shutdown files are executed at logout time. Just like the startup files, we can add commands to clean up the environment when we log out.

Korn Shell:

The Korn shell uses the three profile files as described below:

System Profile File: There is one system level profile file, which is stored in the /etc directory. Maintained by the system administrator, it contains general commands and variable settings that are applied to every user of the system at login time. It is generally quiet large and contains many advanced commands. The system profile file is read-only file; its permissions are set so that only the system administrator can change it.

Personal Profile File: The personal profile, ~/.profile contains commands that are used to customize the startup shell. It is an optional file that is run immediately after the system profile file. Although it is a user file, it is often created by the system administrator to customize a new user's shell.

Environment File: In addition, the Korn shell has an environmental file that is run whenever a new shell is started. It contains environmental variables that are to be exported to subshells and programs that run under the shell.

The environment file does not have a predetermined name. We can give it any name we desire. It must be stored in home directory or in a subdirectory below the home directory. But it is recommended to store in the home directory.

To locate the environmental file, the Korn shell requires that it's absolute or relative pathname be stored in the predefined variable, ENV.

Bash Shell:

For the system profile file, the Bash shell uses the same file as the Korn shell (/etc/profile). However for the personal profile file, it uses one of the three files. First it looks for Bash profile file (~/.bash_profile). If it doesn't find a profile file, it looks for a login file (~/.bash_login).

If it does not find a login file, it looks for a generic profile file (~/.profile). Whichever file the Bash finds, it executes it and ignores the rest. The Bash environmental file uses the same concept as the Korn shell, except that the filename is stored in the BASH_ENV variable.

The Korn shell does not have logout file, but the Bash shell does. When the shell terminates, it looks for the logout file (~/.bash_logout) and executes it.

C Shell:

The C shell uses both startup and shutdown files: it has two startup files, ~/.login and ~/.cshrc, and one shutdown file, ~/.logout.

Login File: The C Shell login file (~/.login) is the equivalent of the Korn and Bash user Profile file. It contains commands and variables that are executed when the user logs in. It is not exported to other shells nor is it executed if the C shell is started from another shell as a subshell.

Environmental File: The C shell equivalent of the environmental file is the ~/.cshrc file. It contains environmental settings that are to be exported to subshells. As an environmental file, it is executed whenever a new subshell is invoked.

Logout File: The C shell logout file ~/.logout, is run when we log out of the C shell. It contains commands and programs that are to be run at logout time.

Other C Shell Files: The C shell may have other system files that are executed at login and logout time. They found in the /etc directory as /etc/csh.cshrc, /etc/csh.login and /etc/csh.logout.

FILTERS AND PIPES – RELATED COMMANDS:

FILTERS:

In UNIX, a filter is any command that gets its input from the standard input stream, manipulates the input and then sends the result to the standard output stream. Some filters can receive data directly from a file.

We have already seen one filter, the **more** command. There are 12 more simple filters available. Three filters – **grep, sed** and **awk** – are so powerful. The following table summarizes the common filters:

FILTER	ACTION
More	Passes all data from input to output, with pauses at the end of the each screen of data.

FILTER	ACTION
Cat	Passes all data from input to output.
Cmp	Compares two files.
Comm	Identifies common lines in two files.
Cut	Passes only specified columns.
Diff	Identifies differences between two files or between common files in two directories.
Head	Passes the number of specified lines at the beginning of the data.
Paste	Combines columns.
Sort	Arranges the data in sequence.
Tail	Passes the number of specified lines at the end of the data.
Tr	Translates one or more characters as specified.
Uniq	Deletes duplicate (repeated) lines.
Wc	Counts characters, words, or lines.
Grep	Passes only specified lines.
Sed	Passes edited lines.
Awk	Passes edited lines – parses lines.

FILTERS AND PIPES:

Filters work naturally with pipes. Because a filter can send its output to the monitor, it can be used on the left of a pipe; because a filter can receive its input from the keyboard, it can be used on the right of a pipe.

In other words a filter can be used on the left of a pipe, between two pipes, and on the right of the pipe. These relationships are presented in figure 2.3:

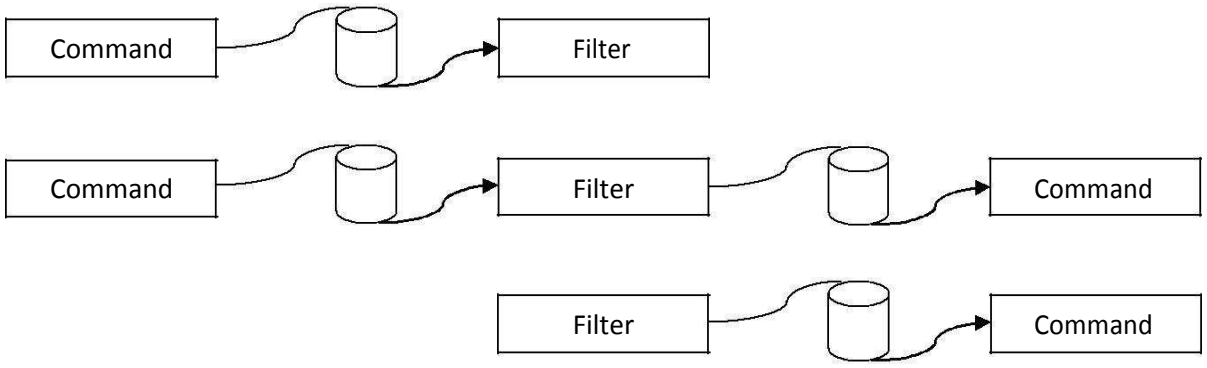
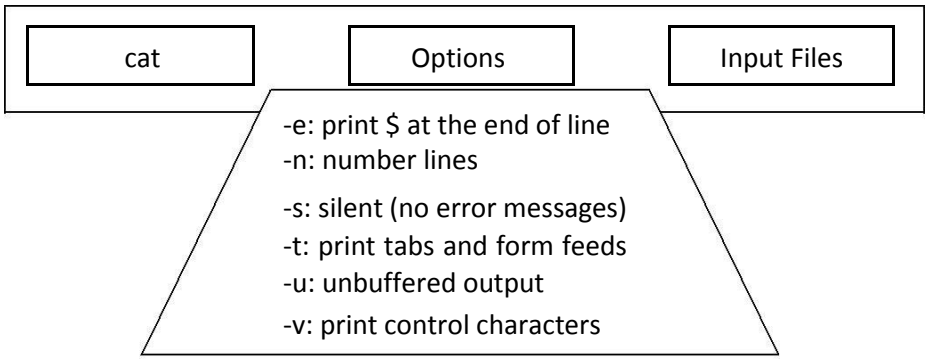


Figure 2.3: Using Filters and Pipes

CONCATENATING FILES:

UNIX provides a powerful utility to concatenate commands. It is known as the catenate command, or **cat** for short. It combines one or more files by appending them in the order they are listed in the command. The input can come from the keyboard; the output goes to the monitor.

The basic concept is shown as follows:



Catenate (cat) command:

Given one or more input files, the **cat** command writes them one after another to standard output. The result is that all of the input files are combined and become one output. If the output file is to be saved, standard output can be redirected to a specified output file. The basic **cat** command as shown below:

cat file1 file2

Example: In the below example if display the contents of three files each of which has only one line.

$ **cat file1 file2 file3**

This is file1.
This is file2.
This is file3.

Using cat to Display a File:

Its basic design makes **cat** a useful tool to display a file. When only one input is provided, the file is catenated with a null file. The result is that the input file is displayed on the monitor. The use of cat command to display a file is as shown below:

cat file1

The following example demonstrates the use of **cat** to display a file. The file, TheRavenV1 contains the first six lines of “TheRaven”.

$ **cat TheRavenV1**

Once up on a midnight dreary, while I pondered, weak and weary, Over many a quaint and curious volume of forgotten lore

While I nodded, nearly napping, suddenly there came a tapping, As of someone gently rapping, rapping at my chamber door.

"'Tis some visitor," I muttered, "tapping at my chamber door Only this and nothing more"

Using cat to create a File:

The second special application uses **cat** to create a file. Again there is only one input this time, however, the input comes from the keyboard. Because we want to save the file, we redirect standard output to a file rather than to the monitor.

Because all of the input is coming from keyboard, we need some way to tell the **cat** command that we have finished inputting data. In other words we need to tell the system that we have input all of the data and are at the end of the file. In UNIX, the keyboard command for **end of file** is the ctrl+d keys, usually abbreviated as ^d.

The following example demonstrates the **cat** command.

$ **cat > goodStudents**

Now is the time
For all good students
To come to the aid of
their college.

cat Options:

There are six options available with **cat**. They can be grouped into four categories: visual characters, buffered output, missing files, and numbered lines.

Visual Characters: Sometimes when we display output, we need to see all of the characters. If the file contains unprintable characters, such as ASCII control characters, we can't see them. Another problem arises if there are space characters at the end of a line – we can't see them because they have no visual graphic.

The visual option **–v** allows us to see control characters, with the exception of the tab, new line, and form feed characters. Unfortunately the way they are printed is not intuitive and is beyond the scope of this text.

If we use the option **–ve** a dollar sign is printed at the end of the each line. If we use the option **–vt**, the tabs appear as ^I. with both options nonprintable characters are prefixed with a caret (^).

Example:

$ **cat –vet catExample**

There is a tab between the numbers on the next line$
1^I2^I3^I4^I5$
$
One two buckle my shoe$

Buffered Output:

When output is buffered, it is kept in the computer until the system has time to write it to a file. Normally **cat** output is buffered. You can force the output to be written to the file immediately by specifying the option –u for unbuffered. This will slows down the system

Missing Files:

When you catenate several files together, if one of them is missing, the system displays a message such as:

Cannot open x.dat: No such file or directory

If you don't want to have this message in your output, you can specify that the **cat** is to be **silent** when it can't find the file. This option is **–s**.

Numbered Lines:

The numbered lines option (-n) numbers each line in each file as the line is written to standard output. If more than one file is being written, the numbering restarts with each file.

Example:

$ **cat –n goodStudents catExample**

1: Now is the time
2: For all good students
3: To come to the aid
4: of their college.

1: There is a tab between the numbers on the next line
2: 1^I2^I3^I4^I5

DISPLAYING BEGINNING AND END OF FILES:

UNIX provides two commands, **head** and **tail** to display the portions of files.

head Command

While the **cat** command copies entire files, the **head** command copies a specified number of lines from the beginning of one or more files to the standard output stream. If no files are specified it gets the lines from standard input. The basic format is given below:

head options inputfiles . . .

The option –N is used to specify the number of lines. If the number of lines is omitted, head assumes 10 lines. If the number of lines is larger than the total number of lines in the file, the total file is used.

Example:

$ **head -2 goodStudents**

Now is the time
For all good students

When multiple files are included in one **head** command, **head** displays the name of file before its output.

Example:

$ **head -2 goodStudents TheRaven**

= => goodStudents <=
= Now is the time
For all good students

= => TheRaven <= =
Once up on a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore

tail Command:

The tail command also outputs data, only this time from the end of the file. The general format of tail command is **tail options inputfile**

Although only file can be referenced (in most systems), it has several options as shown in the table below:

Option	Code	Description
Count from beginning	+N	Skip N-1 lines
Count form end	-N	N lines from end
Count by lines	-l	Measured by lines (default)
Count by characters	-c	Measured by characters
Count by blocks	-b	Measured by blocks
Reverse order	-r	Output in reverse order

Example:

$ **tail -2r**
goodStudents of
their college.
To come to the aid

We can combine head and tail commands to extract lines from the center of a file.

Example:

$ **head -1 TheRaven | tail +2**

Cut and Paste:

In UNIX **cut** command removes the columns of data from a file, and **paste** combines columns of data.

Because the **cut** and **paste** commands work on columns, text files don't work well. Rather we need a data file that organizes data with several related elements on each line. To demonstrate the commands, we created a file that contains selected data on largest five cities in United States according to 1990 census which is as shown in the table below:

Chicago	IL	2783726	3005072	1434029
Houston	TX	1630553	1595138	1049300
Los Angeles	CA	3485398	2968528	1791011
New York	NY	7322564	7071639	3314000
Philadelphia	PA	1585577	1688510	1736895

cut Command:

The basic purpose of cut command is to extract on or more columns of data from either standard input or from one or more file. The format of **cut** command is as shown below:

cut options file list

Since **cut** command looks for columns, we have some way to specify where the columns are located. This is done with one of two command options. We can specify what we want to extract based on character positions with a line or by a field number.

Specifying Character Positions:

Character positions work well when the data are aligned in fixed columns. The data in above table are organized in this way. City is string of 15 characters state is 3 characters (including trailing space), 1990 population is 8 characters, 1980 population is 8 characters and work force is 8 characters.

To specify that file is formatted with fixed columns, we use the character option **–c** followed by one or more column specification. A **column specification** can be one column or a range of columns in the format N-M, where N is the start column and M is the end column, inclusively. Multiple columns are separated by commas.

Example:

$ **cut –c1-14,19-25 censusFixed**

Chicago	2783726
Houston	1630553
Los Angeles	3485398
New York	7322564
Philadelphia	1585577

Field Specification:

While the column specification works well when the data are organized around fixed columns, it doesn't work in other situations. In the table below the city name ranges between columns 1-7 and columns 1-12. Our only choice therefore is to use delimited fields. We have indicated the locations of the tabs with the notation <tab> and have spaced the data to show how it would be displayed.

Chicago	IL<tab>	2783726<tab>	3005072<tab>	1434029
Houston	TX<tab>	1630553<tab>	1595138<tab>	1049300
Los Angeles	CA<tab>	3485398<tab>	2968528<tab>	1791011
New York	NY<tab>	7322564<tab>	7071639<tab>	3314000
Philadelphia	PA<tab>	1585577<tab>	1688510<tab>	736895

When the data are separated by tabs, it is easier to use fields to extract the data from the file. Fields are separated from each other by a terminating character known as a **delimiter.** Any character may be a delimiter; however, if no delimiter **cut** assumes it as a tab character.

To specify a field we use the **field** option (-f). Fields are numbered from the beginning of the line with the first field being field number one. Like the character option, multiple fields are separated by commas with no space after the comma. Consecutive fields may be specified as a range.

The **cut** command assumes that the delimiter is a tab. If it is not we must specify it in the delimiter option. When the delimiter has special meaning to UNIX, it must be enclosed in quotes. Because the space terminates an option, therefore we must enclose it in quotes.

The options of cut command are shown in the table below:

Option	**Code**	**Results**
Character	-c	Extracts fixed columns specified by column number
Field	-f	Extracts delimited columns
Delimiter	-d	Specifies delimiter if not tab (default)
Suppress	-s	Suppresses output if no delimiter in line.

Paste Command:

The paste command combines lines together. It gets the input from two or more files. To specify that the input is coming from the standard input stream, you use a hyphen (-) instead of a filename. The paste command is represented as follows:

paste options file list

The paste combines the first line of the first file with the first line of the second file and writes the combined line to the standard output stream. Between the columns, it writes a tab. At the end of the last column, it writes a newline character. It then combines the next two lines and writes them, continuing until all the lines have been written to standard stream. In other words paste treats each line of each file as a column.

Note: *The cat and paste command are similar: The cat command combines files vertically (by lines). The paste command combines files horizontally (by columns).*

Sorting:

When dealing with data especially a lot of data we need to organize them for analysis and efficient processing. One of the simplest and most powerful organizing techniques is sorting. When we sort data we arrange them in sequence.

Usually we use ascending sequence, an arrangement in which each piece of data is larger than its predecessor. We can also sort in descending sequence, in which each piece of data is smaller than its predecessor.

sort Command:

The sort utility uses options, field specifiers, and input files. The filed specifiers tell it which fields to use for the sort. The sort command format is shown below:

sort options field specifiers input files

Sort by lines:

The easiest sort arranges the data by lines. Starting at the beginning of the line, it compares the first character in one line with the first character in another line. If they are the same, it moves to the second character and compares them. This character-by-character comparison continues until either all character in both lines have compared equal or until two unequal characters are found.

If lines are not equal, comparison stops and sort determines which line should be first based on the two unequal characters. In comparing characters, sort uses the ASCII values of each character.

Field specifiers:

When a field sort is required, we need to define which field or fields are to be used for the sort. Field specifiers are a set of two numbers that together identify the first and last field in a sort key. They have the following format:

$$+\text{number}_1\ -\text{number}_2$$

number_1 specifies the number of fields to be skipped to get to the beginning of the sort field, whereas number_2 specifies the number of fields to be skipped, relative to the beginning of the line to get to the end of the sort key.

TRANSLATING CHARACTERS:

There are many reasons for translating characters from one set to another. One of the most common is to convert lowercase characters to uppercase, or vice versa. UNIX provides translate utility making conversions from one set to another.

tr command: The tr command replaces each character in a user-specified set of characters with a corresponding character in a second specified set. Each set is specified as a string. The first character in the first set is replaced by the first character in the second set; the second character in the first set is replaced by the second character in the second set and so forth until all matching characters have been replaced. The strings are specified using quotes.

The tr command is represented as follows: **tr options string1 string2**

Simple Translate:

Translate receives its input from standard input and writes its output to standard output. If no options are specified, the text is matched against the string1 set, and any matching characters are replaced with the corresponding characters in the string2 set. Unmatched characters are unchanged.

$ **tr "aeiou" "AEIOU"**

It is very easy to use TRANSLATE. #input It Is
vEry EAsy tO UsE TRANSLATE. #output

Nonmatching Translate Strings:

When the translate strings are of different length, the result depends on which string is shorter. If string2 is shorter, the unmatched characters will all be changed to the last character in string2. On the other hand, if string1 is shorter, the extra characters in string2 are ignored.

$ **tr "aeiou" "AE?"** #case 1: string2 is shorter than string1 It is very
easy to use TRANSLATE.
It ?s vEry EAsy t? ?sE trAnslAtE.

$ **tr "aei" "AEIOU?"** #case 1: string1 is shorter than string2 It is very
easy to use TRANSLATE.
It Is vEry EAsy to usE trAnslAtE.

Delete Characters:

To delete matching characters in the translation we use the delete option (-d). In the following example we delete all vowels, both upper and lowercase. Note that the delete option does not use string2.

$ **tr –d "aeiouAEIOU"**

It is very easy to use
TRANSLATE t s vry sy t s TRNSLT

Squeeze Output:

The squeeze option deletes consecutive occurrences of the same character in the output. **Example:**

$ **tr –s "ie" "dd"**

The fiend did dastardly deeds
Thd fdnd d dastardly ds

Complement:

The complement option reverses the meaning of the first string. Rather than specifying what characters are to be changed, it says what characters are not to be changed.

Example:

$ **tr –c "aeiou" "*"**
It is very easy to use TRANSLATE.
ie***ea****o*u*e**********

FILES WITH DUPLICATE LINES:

We used a sort to delete duplicate lines (words). If the lines are already adjacent to each other, we can use the unique utility.

uniq command:

The **uniq** command deletes duplicate lines, keeping the first and deleting the others. To be deleted, the lines must be adjacent. Duplicate lines that are not adjacent are not deleted. To delete nonadjacent lines the file must be sorted.

Unless otherwise specified the whole line can be used for comparison. Options provide for the compare to start with a specified field or character. The compare whether line, field, or character, is to the end of the line. It is not possible to compare one filed in the middle of the line. The unique command is shown as follows:

uniq options inputfile

All of the examples use a file with three sets of duplicate lines. The complete file is shown below:

```
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
5 completely duplicate lines
Not a duplicate - - next duplicates first 5
5 completely duplicate lines
Last 3 fields duplicate: one two three
Last 3 fields duplicate: one two three
```

Last 3 fields duplicate: one two three The
next 3 lines are duplicate after char 5
abcde Duplicate to end
fghij Duplicate to end
klmno Duplicate to end

There are three options: output format, skip leading fields and skip leading characters.

Output Format:

There are four output formats: nonduplicated lines and the first line of each duplicate series (default), only unique lines (-u), only duplicated lines (-d), and show count of duplicated lines (-c).

Default Output Format:

We use unique command without any options. It writes all of the nonduplicated lines and the first of a series of duplicated lines. This is the default result.

$ **uniq uniqFile**

5 completely duplicate lines

Not a duplicate - - next duplicates first 5
5 completely duplicate lines

Last 3 fields duplicate: one two three The
next 3 lines are duplicate after char 5
abcde Duplicate to end

fghij Duplicate to end
klmno Duplicate to end

Nonduplicated Lines (-u):

The nonduplicated lines option is –u. It suppresses the output of the duplicated lines and lists only the *unique lines in the file*. Its output is shown is the example below:

$ **uniq –u uniqFile**

Not a duplicate - - next duplicates first 5
5 completely duplicate lines

The next 3 lines are duplicate after char 5
abcde Duplicate to end

fghij Duplicate to end
klmno Duplicate to end

Only Duplicated Lines (-d):

The opposite of nonduplicated lines is to write only the duplicated lines (-d). Its output is shown in the example below:

$ **uniq –d uniqFile**

5 completely duplicate lines
Last 3 fields duplicate: one two three

Count Duplicate Lines (-c):

The count duplicates option (-c) writes all of the lines, suppressing the duplicates, with a count of number duplicates at the beginning of the line.

$ **uniq –c uniqFile**

5 5 completely duplicate lines
1 Not a duplicate - - next duplicates first 5
1 5 completely duplicate lines
3 Last 3 fields duplicate: one two three
1 The next 3 lines are duplicate after char 5
1 abcde Duplicate to end
1 fghij Duplicate to end
1 klmno Duplicate to end

Skip Leading Fields:

While the default compares the whole line to determine if two lines are duplicate, we can also specify where the compare is to begin. The skip duplicate fields option (-f) skips the number of fields specified starting at the beginning of the line and any spaces between them. Remember that a field is defined as a series of ASCII characters separated by either a space or by a tab. Two consecutive spaces would be two fields; that's the reason **uniq** skips leading spaces between fields.

Example:

$ **uniq –d –f 4 uniqFile**

5 completely duplicate lines
Last 3 fields duplicate: one two three
abcde Duplicate to end

Skipping Leading Characters:

We can also specify the number of characters that are to be skipped before starting the compare. In the following example, note that the number of leading characters to be skipped is separated from the option (-s). This option is represented as in below example:

$ **uniq –d –s 5 uniqFile**

5 completely duplicate lines

Last 3 fields duplicate: one two
three abcde Duplicate to end

The unique options are represented as in the table below:

Option[a]	**Code**	**Results**
Unique	-u	Only unique lines are output.
Duplicate	-d	Only duplicate lines are output.
Count	-c	Outputs all lines with duplicate count.
Skip field	-f	Skips leading fields before duplicate test.
Skip characters	-s	Skips leading characters before duplicate test.

COUNT CHARACTERS, WORDS, OR LINES:

Many situations arise in which we need to know how many words or lines are in a file. Although not as common, there are also situations in which we need to know a character count. The UNIX word count utility handles these situations easily.

wc command:

The **wc** command counts the number of characters, words, and lines in one or more documents. The character count includes newlines (/n). Options can be used to limit the output to only one or two of the counts. The word count format is shown below:

wc options inputfiles

The following example demonstrates several common count combinations. The default is all three options (clw). If one option is specified, the other counts are not displayed. If two are specified the third count is not displayed.

$ **wc TheRaven**

116 994 5782 TheRaven

$ **wc TheRaven uniqFile**

```
 116   994   5782 TheRaven
  14    72    445 uniqFile
 130  1066   6227 total
```

$ **wc –c TheRaven**

```
5782 TheRaven
```

$ **wc –l TheRaven**

```
116 TheRaven
```

$ **wc –cl TheRaven**

```
 116  5782 TheRaven
```

The following table shows the word count options:

Option	**Code**	**Results**
Character count	-c	Counts characters in each file.
Line count	-l	Counts number of lines in each file.
Word count	-w	Counts number of words in each file.

COMPARING FILES:

There are three UNIX commands that can be used to compare the contents of two files: compare (**cmp**), difference (**diff**), and common (**comm**).

Compare (cmp) Command:

The **cmp** command examines two files byte by byte. The action it takes depends on the option code used. Its operation is shown below:

cmp options file1 file2

cmp without Options:

When the **cmp** command is executed without any options, it stops at the first byte that is different. The byte number of the first difference is reported. The following example demonstrates the basic operation, first with two identical files and then with different files.

$ **cat cmpFile1**

```
123456
7890
```

$ **cat cmpFile1.cpy**
123456
7890

$ **cmp cmpFile1 cmpFIle1.cpy**
$ **cat cmpFile2**
123456
As9u

$ **cmp cmpFile1 cmpFile2**
cmpFile1 cmpFile2 differ: char 8, line2

cmp with List Option (-l)

The list option displays all of the differences found in the files, byte by byte. A sample output is shown in the following example:

$ **cmp –l cmpFIle1 cmpFile2**

8	67	141
9	70	163
11	60	165

cmp with suppress list option (-s):

The suppress list option (-s) is similar to the default except that no output is displayed. It is generally used when writing scripts. When no output is displayed, the results can be determined by testing the exit status. If the exit status is 0, the two files are identical. If it is 1, there is at least one byte that is different. To show the exit status, we use the **echo** command.

$ **cmp cmpFile1**
cmpFile1.cpy $ **echo $?**
0
$ **cmp cmpFile1 cmpFile2**
$ **echo $?**
1

Difference (diff) Command:

The **diff** command shows the line-by-line difference between the two files. The first file is compared to the second file. The differences are identified such that the first file could be modified to make it match the second file.

The command format is shown below:

diff options files or directories

The diff command always works on files. The arguments can be two files, a file and directory, or two directories. When one file and one directory are specified, the utility looks for a file with the same name in the specified directory. If two directories are provided, all files with matching names in each directory are used. Each difference is displayed using the following format:

range1 action **range2**
< text from file1
- - -
> text from file2

The first line defines what should be done at range1 in file1 (the file identified by first argument) to make it match the lines at range2 in file2 (the file identified by second argument). If the range spans multiple lines, there will be a text entry for each line in the specified range. The action can be change (c), append (a), or delete (d).

Change (c) indicates what action should be taken to make file1 the same as file2. Note that a change is a delete and replace. The line(s) in range1 are replaced by the line(s) in range2.

Append (a) indicates what lines need to be added to file1 to make it the same as file2. Appends can take place only at the end of the file1; they occur only when file1 is shorter than file2.

Delete (d) indicates what lines must be deleted from file1 to make it the same as file2. Deletes can occur only if file1 is longer than file2.

Example:	**Interpretation**
6c6	change: Replace line 6 in file1 with line 6 in file2
< hello	
- - -	
> greeting	
25a26,27	append: At the end of fil1 (after line 25), insert 26 and 27 from file2.
> bye bye.	Note that for append, there is no separator (dash) and no file1 (<) lines.
> good bye.	
78,79d77	delete: the extra lines at the end of file1 should be deleted.
< line 78	The text of the lines to be deleted is shown.
text	Note again that there is no separator line
< line 79	and, in this case no file2 (>) lines.

Common (comm) Command:

The **comm** command finds lines that are identical in two files. It compares the files line by line and displays the results in three columns. The left column contains unique lines in file 1; the center column contains unique lines in file 2; and the right column contains lines found in both files. the command format is shown below:

comm options file1 file2

The files (comm1 and comm2) used to demonstrate the **comm** command is shown in the table given below:

comm1	comm2
one same	one same
two same	two same
different comm1	different comm2
same at line 4	same at line 4
same at line 5	same at line 5
not in comm2	
same at line 7	same at line 7
same at line 8	same at line 8
not in comm1	
last line same	last line same

The output for comm utility for these two files is shown below:

$ **comm comm1 comm2**

```
                    one same
                    two same
different comm1
          different comm2
                    same at line 4
                    same at line 5
not in comm2
                    same at line 7
                    same at line 8
          not in comm1
                    last line same
```

UNIT-III

UNIT-III

COMMUNICATIONS:

UNIX provides a rather rich set of communication tools. Even with all of its capabilities, if you see email extensively, you will most likely want to use a commercial product to handle your mail.

The communication utilities in UNIX are: ***talk, write, mail, Telnet, FTP***

USER COMMUNICATION:

The first two communication utilities are **talk** and **write,** deal with communications between two users at different terminals.

talk command:

The **talk** command allows two UNIX users to chart with each other just like you might do on the phone except that you type rather than talk. When one user wants to talk to another, he or she simply types the command **talk** and the others person's login id. The command format is shown below:

talk options user_id terminal

The conversation doesn't begin however until the called user answers. When you send a request to talk, the user you are calling gets a message saying that you want to talk. The message is shown here:

Message from Talk_Daemon@challenger at 18:07 . . .

talk: connection requested by gilberg@challenger.atc.fhda.edu.
talk: respond with: talk gilberg@challenger.atc.fhda.edu.

It your friend doesn't want to talk, he or she can ignore the message, much like you don't have to answer the phone when it rings. However UNIX is persistent, it will keep repeating the message so that the person you are calling has to respond.

There are two possible responses. The first which agrees to accept the call is a corresponding **talk** command to connect to you. This response is seen in the third line of the preceding example. Note that all that is needed is the user id. The address starting with the at sing (@) is not required if the person calling is on the same system.

To refuse to talk, the person being called must turn messages off. This is done with the message (**mesg**) command as shown below:

mesg n

The message command has one argument, either y, yes I want to receive messages, or n, no I don't want to receive messages. It is normally set to receive messages when you log in to the system. Once you turn it off, it remains off until you turn it back on or restart. To turn it on, set it to yes as follow:

mesg y

To determine the current message status key **mesg** with no parameters as shown in the example below (the response is either yes (y) or no (n)):

$ **mesg**

is n

when you try to talk with someone who has messages turned off, you get the following message:

[Your party is refusing messages]

After you enter the talk response the screen is split in to two portions. In the upper portion which represents your half of the conversation, you will see a message that the person you are calling is being notified that you want to talk. This message is:

[Waiting for your party to respond]

Once the connection has been made you both see a message saying that the connection has been established. You can then begin talking. What you type is shown on the top half of the screen. Whatever you type is immediately shown in the bottom half of your friend's screen.

write command:

The **write** command is used to send a message to another terminal. Like **talk**, it requires that the receiving party be logged on. The major difference between write and talk is that write is one way transmission. There is no split screen, and the person you are communicating with does not see the message as it is being typed. Rather it is sent one line at a time; that is the text is collected until you key Enter, and then it is all sent at once.

You can type as many lines as you need to complete your message. You terminate the message with either an end of file (ctrl+d) or a cancel command (ctrl+c). When you terminate the message, the recipient receives the last line and end of transmission (<EOT>) to indicate that the transmission is complete. The format for the **write** is shown below:

When you write to another user, a message is sent telling him or her that you are about to send a message. A typical message follows. It shows the name of the sender, the system the message is coming from, the sender's terminal id, and the date and time.

Message from Joan on sys (ttyq1) [Thu Apr 9 21:21:25]

Unless you are very quick, the user can quickly turn your message off by keying *mesg n*. when this happens, you get the following error when you try to send your message:

Can no longer write to /dev/ttyq2

If you try to write to a user who is not logged on, you get the following error message:

dick is not logged on.

Sometimes a user is logged on to more than one session. In this case UNIX warns you that he or she is logged on multiple times and shows you all of the alternate sessions. A sample of this message is:

Joan is logged on more than one place.
You are connected to "ttyq1".
Other locations
are: ttyq2

ELECTRONIC MAIL:

Although talk and write are fastest ways to communicate with someone who's online, it doesn't work it they're not logged on. Similarly if you need to send a message to someone who's on a different operating system, they don't work.

There are many different email systems in use today. While UNIX does not have all of the capabilities of many of the shareware and commercial products, it does offer a good basic system.

A mail message is composed of two parts: the header and the body. The **body** contains the text of the message. The **header** consists of the subject, the addressees (To:), sender (From:), a list of open copy recipients (Cc:) and a list of blind copy recipients (Bcc:).

In addition to the four basic header entries there are seven extended fields:

1) Reply-To: if your do not specify a reply-to-address, your UNIX address will automatically be used.

2) Return-Receipt-To: if the addressee's mail system supports return recipiets, message will be sent to the address in return-receipt-to when the message is delivered to the addressee.

3) In-Reply-To: this is a text string that you may use to tell the user you are replying to a specific note. Only one line is allowed. It is displayed as header data when the addressee opens the mail.

4) References: when you want to put a reference, such as to a previous memo sent by the recipient you can include a references field.

5) Keywords: Provides a list of keywords to the context of the message.

6) Comments: allows you to place a comment at the beginning of the message.

7) Encrypted: Message is encrypted.

All of these data print as a part of the header information at the top of the message. A final word of caution on extended header information: Not all email systems support all of them.

Mail Addresses:

Just as with snail mail, to send email to someone, you must know his or her address. When you send mail to people on your own system, their address is their user id. So Joan can send mail to Tran using his id, *tran*. This is possible because *tran* is a local username or alias and UNIX knows the address for everyone on it.

To send mail to people on other systems, however you need to know not only their local address (user id) but also their domain address. The local address and domain address are separated by an at sign (@) as shown below:

Local Address@Domain Address

Local Address:

The local address identifies a specific user in a local mail system. It can be user id, a login name, or an alias. All of them refer to the user mailbox in the directory system. The local address is created by the system administrator or the postmaster.

Domain Address:

The domain address is a hierarchical path with the highest level on the right and the lowest level on the left. There must be at least two levels in the domain address.

The parts of the address are separated by dots (periods). The highest level is an internet label. When an organization or other entity joins the internet, it selects its internet label from one of the two label designations: generic domain labels or country domain labels. Below the domain label, the organization (system administrator), controls the hierarchical breakdown, with each level is referring to a physical or logical organization of computers or services under its control.

Generic Domains:

Most Internet domains are drawn from the generic domain levels, also known as top-level domains. These generic names represent the basic classification structure for the organizations in the grouping (table given below).

Label	Description
com	Commercial (business) profit organizations
edu	Educational institutions (college or university)
gov	Government organizations at any level.
int	International organizations.
mil	Military organizations
net	Network support organizations such as ISP's (internet service providers)
org	Nonprofit organizations

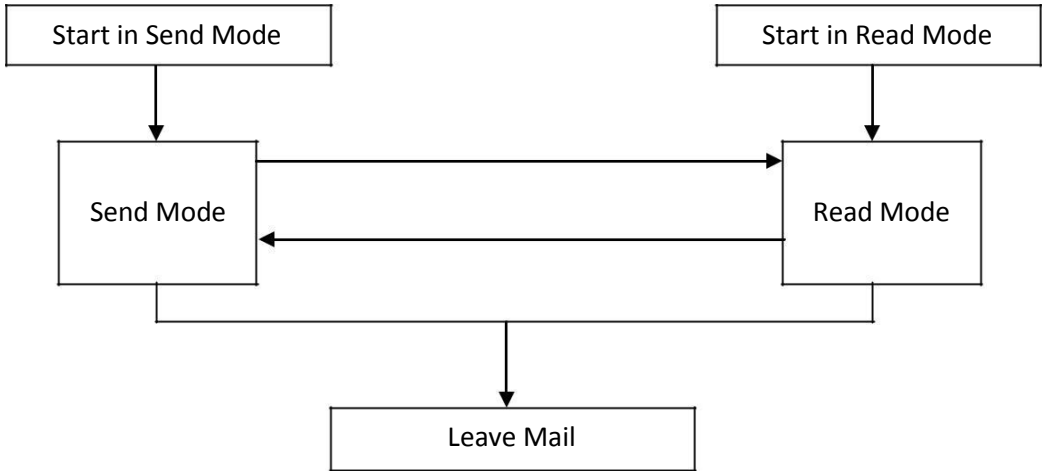
Country Domain:

In the country domain, the domain label is a two-character designation that identifies the country in which the organization resides. While the rest of the domain address varies, it often uses a governmental breakdown such as state and city. A typical country domain address might look like the following address for an internet provider in California:

yourInternet.ca.us

Mail Mode:

When you enter, the mail system, you are either in the send mode or the read mode. When you are in the send mode, you can switch to the read mode to process your incoming mail; likewise, when you are in the read mode, you can switch to the send mode to answer someone's mail. You can switch back and forward as often as necessary, but you can be in only one mode at a time. The basic mail system operation appears in the figure as follows:



mail Command:

The mail command is used to both read and send mail. It contains a limited text editor for composing mail notes.

Send Mail:

To send mail from the system prompt, you use the mail command with one or more addresses as shown below:

mail options address(es)

example: $ **mail tran, dilbert, harry, carolyn@atc.com**

When your message is complete, it is placed in the mail system spool where the email system can process it. If you are sending the message to more than one person, their names may be separated by commas or simply by spaces.

If the person you are sending mail to is not on your system, you need to include the full email address. The last addressee (carolyn@atc.com) is a person on a different system. *The mail system provides a very limited text editor for you to write you note. It does not text wrap, and once you have gone to the next line, you cannot correct previous lines without invoking a text editor.*

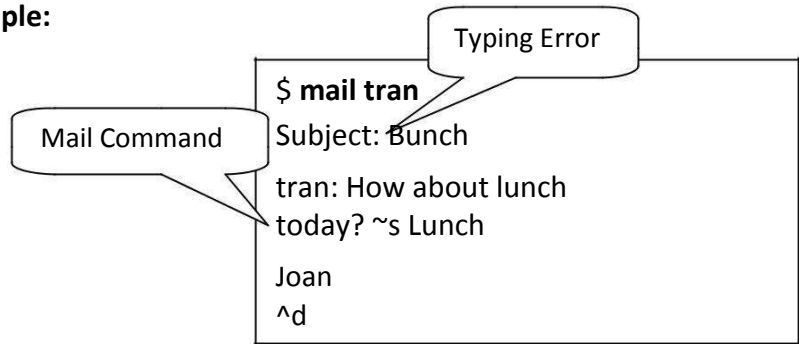
To invoke the editor use the edit message command (~e). This will place you in your standard editor such as vi.

Once in the editor you can use any of its features. When you are through composing or correcting your message, the standard save and exit command will take you back to your send mail session.

Send Mail Commands:

The mail utility program allows you to enter mail commands as well as text. It assumes that you are entering text. To enter a mail command, you must be at the beginning of the line; a mail command in the middle of a line is considered text. When the first character you enter on the line is a tilde (~), the mail utility interprets the line as a command.

Example:



In the above figure, Joan makes a typing mistake and doesn't notice it until she is ready to send the message. Fortunately there is mail command to change the subject line. So she simply types ***~s Lunch*** at the beginning of a line, and when Tran receives the message, it has a correct subject line. The complete list of send mail commands is given the table below:

Command	Action
~~	Quotes a single tilde.
~b users	Appends users to bcc ("blind" cc) header field.
~cm text	Appends users to cc header field.
~d	Reads in dead.letter.
~E or ~eh	edits the entire message
~e	edits the message body
~en text	Puts text in encrypted header field.
~f messages	Reads in messages.
~H	Prompts for all possible header fields.
~h	Prompts for important header fields.
~irt text	Appends text to in-response-to header field.
~k text	Appends text to keywords header field.
~m messages	Reads in messages, right shifted by a tab.
~p	prints the message buffer
~q	quits: do not send
~r file	reads a file in to the message buffer
~rf text	Appends text to references header field.
~rr	Toggles Return-Receipt-To header field
~rt users	Appends users to Reply-To header field
~s text	Puts text in subject header field.
~t users	Appends users to To header field.

Quit mail and write command:

To quit the mail utility, you enter ctrl+d. If you are creating a message, it is automatically sent when you enter ctrl+d. If you are not ready to send the message and must still quit, you can save it in a file. To write the messages to a file, you use the **write** command (~w). The name of the file follows the command and once the file has been written, you exit the system using the **quit** command (~q) as shown in the example below:

~w noteToTran
q

To quit without saving file you use the quit command. In truth the quit command saves your mail in a special dead-letter file known as ***dead.letter***. You can retrieve this file and continue working on it at a later date just like any other piece of mail you may have saved.

Reloading Files: The Read File Command:

There are two basic reasons for reading a file in to a message. First if you have saved a file and want to continue working on it, you must reload it to the send mail buffer. The second reason is to include a standard message or a copy of a note you received from someone else.

To copy a file in to a note, open mail in the send mode as normal and then load the saved file using the read file command (~r). When you load the file, however the file is not immediately loaded. Rather the mail utility simply makes a note that the file is to be included.

Example:

$ **mail tran**
Subject: Lunch
~r note-to-tran
"note-to-tran" 3/52

To see the note and continue working on it, you use the print command (~p). This command reprints the mail buffer, expanding any files so that you can read their contents. If the first part of the message is okay, you can simply continue entering the text.

If you want to revise any of your previous work, then you need to use an editor. To start an editor use either the start editor (~e) or start vi command (~v). The start editor command starts the editor designated for mail.

If none is specified, vi is automatically used. When you close the editor you are automatically back in the mail utility and can then send the message or continue entering more text.

Distribution Lists:

Distribution Lists are special mail files that can contain aliases for one or more mail addresses. For example you can create a mail alias for a friend using only his or her first name.

Then you don't need to key a long address that may contain a cryptic user id rather than a given name. Similarly if you are working on a project with 15 others, you can create a mail distribution list and use it rather than typing everyone's address separately.

Distribution lists are kept in a special mail file known as *.mailrc*. Each entry in the distribution list file begins with alias and designates one or more mail addresses. Because all entries start with the word alias, they are known as alias entries.

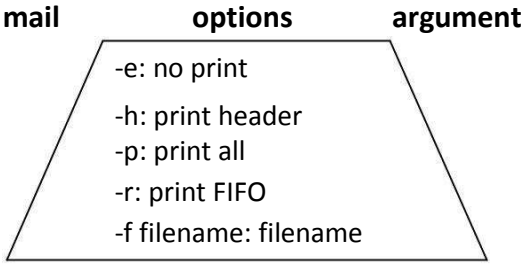
To add an alias entry to the distribution list you must edit it and insert the alias line in the following format:

alias project joan jxc5936 tran bill@testing.atc.com

In this example the alias name is "project". Within project we have included four mail addresses. The first three are for people on our system as indicated by the local names. The last is for a person on a different system.

Read Mode:

The read mode is used to read your mail. Its operation is presented below:



Although the mail system is a standard utility its implementation is anything but standard. You enter mail in the read mode by simply keying mail at the command line with-out any parameters (addresses). If you don't have any mail a no-mail message is printed and you stay at the command line. A typical no-mail response is as follows:

$ **mail**
No mail for tran

Reading New Mail: If you receive notice that mail has arrived while you are in the read mail mode, you will not be able to read it until you redisplay the mail list.

If you use the headers command to reprint the list, you will still not see the new mail. To reprint the list with the new header, you must use the folder command (folder %) at the mail prompt.

The folder command may be abbreviated fo %. Note however that this command has the same effect as quitting mail and reentering. For example, if you had deleted any mail, you will not be able to undelete it.

Replying to and Forwarding Mail

After you have read a piece of mail, you can reply to it. There are two reply commands. The first command (R) replies only to the sender. The second command (r) replies to the sender and all of the addressees. Since these two commands are so close, you want to get into the habit of reading the To list when you use reply.

When you use reply, you will see the header information for the To and Subject fields. If you want to add other header information, you must use the tilde commands described in send mail. **Note:** & is the read mail prompt.

Example:

& **r**
To: **joan**
Subject: **re: Project Review**

Quitting:

Quit terminates mail. There are three ways to quit. From the read mail prompt, you can enter the quit command (q), the quit and do not delete command (x), or the end of transmission command (ctrl+d). All undeleted messages are saved in your mailbox file.

Deleted messages, provided they weren't undeleted, are removed when you use quit (q) and end (ctrl+d); they are not deleted when you use do not delete (x). If new mail has arrived while you were in mail, an appropriate message is displayed.

Saving Messages:

To keep you mail organized, you can save messages in different files. For example all project messages can be saved in a project file, and all status reports can be kept in another file. To save a message, use the command (s or s#). The simple save command saves the current message in the designated file as in the following example:

s project

If the file project in this example doesn't exist, it will be created. If it does exist, the current message is appended to the end of the file. An alternative to saving the message is to write it to a file. This is done with the write command (w), which also requires a file. The only difference between a save and write is that the first header line in the mail is not written; the message body of each written message is appended to the specified file.

To read messages in a saved file, start mail with the file option on the command prompt as shown next. Note that there is no space between the option and the filename.

$ **mail -fproject**

Deleting and Undeleting Mail:

Mail remains in your mailbox until you delete it. There are three ways to delete mail. After you have read mail, you may simply key ***d*** at the mail prompt, and it is deleted. Actually anytime you key ***d*** at the mail prompt, the current mail entry is deleted. After you read mail, the mail you just read remains the current mail entry until you move to another message.

If you are sure of the message number you can use it with delete command. For example to delete message 5, simply key ***d5*** at the mail prompt. You can also delete a range of messages by keying a range after the delete command. All three of these delete formats are shown below:

```
& d          # Deletes current mail entry
& d5         # Deletes entry 5 only
& d5..17     # Deletes entries 5 through 17
```

You can undelete a message as long as you do it before you exit mail or use the folder command. To undelete a message you use the undelete command (u) and the message number. To undelete message 5, key u5. You cannot undelete multiple messages with one command.

Read Mail Commands:

The complete list of read mail commands are shown in the table below:

Mail Command	**Explanation**
t <message list>	Types messages.
n	Goes to and types next message.
e <message list>	Edits messages.
d <message list>	Deletes messages.

Mail Command	Explanation
folder % or fo %	Reprints mail list including any mail that arrived after start of read mail session. Has the effect of quitting read mail and reentering.
s <message list> file	Appends messages to file.
u <message list>	Undelete messages
R <message list>	Replies to message senders
r <message list>	Replies to message senders and all recipients
pre <message list>	Makes messages go back to incoming mail file.
m <user list>	Mails to specific users.
h <message list>	Prints out active message headers
q	Quits, saving unresolved messages in mailbox.
x	Quits, do not remove from incoming mail file.
w <number> filename	Appends body of specified message number to filename.
!	Shell escape.
cd [directory]	Changes to directory or home if none given.

Read Mail Options:

There are five read mail options of interest. They are shown in the table below:

Option	Usage
-e	Does not print messages when mail starts.
-h	Displays message header list and prompt for response on start.
-p	Prints all messages on start
-r	Prints messages in first in, first out (FIFO) order.
-f file_name	Opens alternate mail file (file_name).

Mail Files:

UNIX defines two sets of files for storing mail: arriving mail and read mail.

Arriving Mail Files:

The system stores mail in a designated file when it arrives. The absolute path to the user's mail file is stored in the MAIL variable. As it arrives incoming mail is appended to this file. a typical mail path is in the next example:

$ **echo $MAIL**

/usr/mail/forouzan

The mail utility checks the incoming mail file periodically. When it detects that new mail has arrived, it informs the user. The time between mail checks is determined by a system variable, MAILCHECK. The default period is 600 seconds (10 minutes). To change the time between the mail checks we assign a new value, in seconds, to MAILCHECK as follows:

$ **MAILCHECK = 300**

Read Mail File:

When mail has been read, but not deleted, it is stored in the mbox file. As mail is read, it is deleted from the incoming mail file and moved to the mbox file. This file is normally stored in the user's home directory.

REMOTE ACCESS:

telnet → **te**rmina**l net**work

The telnet Concept:

The telnet utility is a TCP/IP standard for the exchange of data between computer systems. The main task of telnet is to provide remote services for users. For example we need to be able to run different application programs at a remote site and create results that can be transferred to our local site.

One way to satisfy these demands is to create different client/server application programs for each desired service. Programs such as file transfer programs and email are already available. But it would be impossible to write a specific client/server program for each requirement.

The better solution is a general-purpose client/server program that lets a user access any application program on a remote computer; in other words, it allows the user to log into a remote computer. After logging on, a user can use the services available on the remote computer and transfer the results back to the local computer.

Time-Sharing Environment:

Designed at a time when most operating systems such as UNIX, were operating in a time-sharing environment, telnet is the standard under which internet systems interconnect. In a time sharing environment a large computer supports multiple users. The interaction between a user and the computer occurs through a terminal, which is usually a combination of keyboard, monitor and mouse. Our personal computer can simulate a terminal with a terminal emulator program.

Login:

In a time sharing environment, users are part of the system with some right to access resources. Each authorized user has identification and probably a password. The user identification defines the user as a part of the system. To access the system, the user logs into the system with a user id or login name. The system also facilitates password checking to prevent an unauthorized user from accessing the resources.

Local Login: When we log in to a local time sharing system, it is called local login. As we type at a terminal or a work station running a terminal emulator, the keystrokes are accepted by the terminal driver. The terminal driver passes the characters to the operating system. The operating system in turn interprets the combination of characters and invokes the desired application program or utility.

Remote Login: When we access an application program or utility located on a remote machine, we must still login only this time it is a remote login. Here the telnet client and server programs come into use. We send the keystrokes to the terminal driver where the local (client) operating system accepts the characters but does not interpret them. The characters are sent to the client telnet interface, which transforms the characters to a universal character set called **Network Virtual Terminal** (NVT) characters and then sends them to the server using the networking protocols software.

The commands or text, in NVT form, travel through the Internet and arrive at the remote system. Here the characters are delivered to the operating system and passed to the telnet server, which changes the characters to the corresponding characters understandable by the remote computer.

However the characters cannot be passed directly to the operating system because the remote operating system is not designed to receive characters from a telnet server: It is designed to receive characters from a terminal driver. The solution is to add a piece of software called a pseudo terminal driver, which pretends that the characters are coming from a terminal. The operating system then passes the characters to the appropriate application program.

Connecting to the Remote Host:

To connect to a remote system, we enter the telnet command at the command line. Once the command has been entered, we are in the telnet system as indicated by the telnet prompt. To connect to a remote system, we enter the domain address for the system. When the connection is made, the remote system presents its login message. After we log in, we can use the remote system as though it were in the same room. When we complete our processing, we log out and are returned to our local system.

There are several telnet subcommands available. The more common ones are listed in table given below:

Command	Meaning
open	Connects to a remote computer
close	Closes the connection
display	Shows the operating parameters
mode	Changes to line mode or character mode
Set	Sets the operating parameters
status	Displays the status information
send	Sends special characters
quit	Exits telnet
?	The help command. telnet displays its command list

FILE TRANSFER:

Whenever a file is transferred from a client to a server, a server to a client or between two servers, a transfer utility is used. In UNIX the ftp utility is used to transfer files.

The ftp Command:

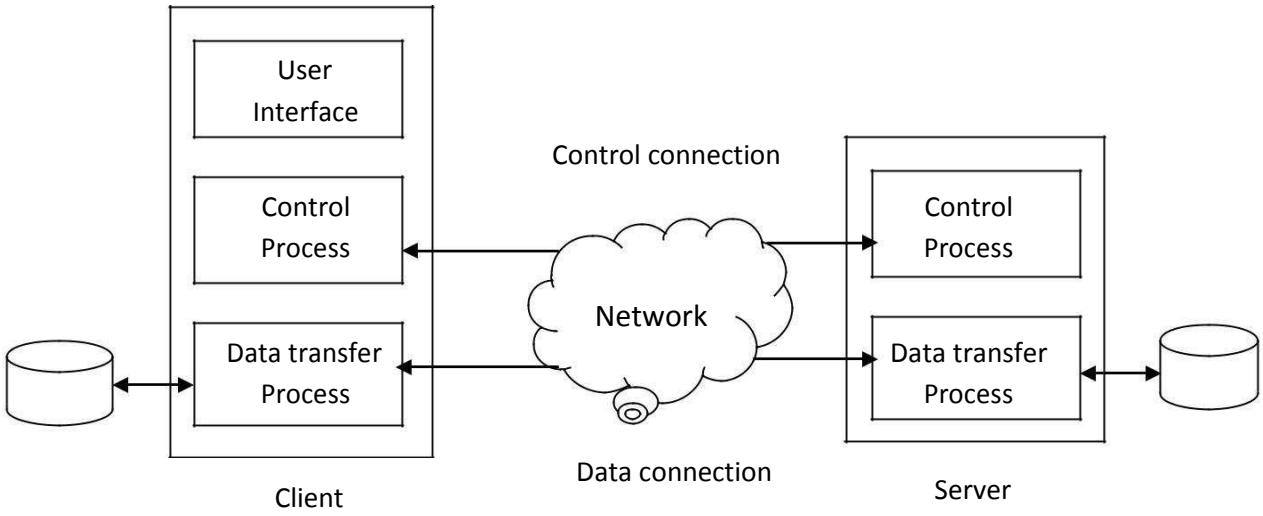
File Transfer Protocol (ftp) is a TCP/IP standard for copying a file from one computer to another. Transferring files from one computer to another is one of the most common tasks expected from a networking or internetworking environment.

The ftp protocol differs from other client-server applications in that it establishes two connections between the hosts. One connection is used for data transfer, the other for control information (commands and responses). Separation of commands and data transfer makes ftp more efficient.

The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred.

The following figure shows the basic ftp mode. The client has three components: user interface, client control process, and the client data transfer process. The server has two components: the server control process and the server data transfer process. The control connection is made between the control processes. The data connection is made between the data transfer processes.

The control connection remains connected during the entire interactive ftp session. The data connection is opened and then closed for each file transferred. It opens each time a file transfer command is used, and it closes when the file has been transferred.



Establishing ftp Connection:

To establish an ftp connection, we enter the ftp command with the remote systems domain name on the prompt line. As the alternative, we can start the ftp session without naming the remote system. In this case we must open the remote system to establish a connection. Within ftp, the open command requires the remote system domain name to make the connection.

Closing an ftp connection:

At the end of the session, we must close the connection. The connection can be closed in two ways: to close the connection and terminate ftp, we use quit. After the connection is terminated, we are in the command line mode as indicated by the command prompt ($).

Example for Terminate the ftp session:

ftp> **quit**
221 Goodbye.
$

To close the connection and leave the ftp active so that we can connect to another system, we use close. We verify that we are still in an ftp session by the ftp prompt. At this point, we could open a new connection.

Example for close the ftp session:

ftp> **quit**
221 Goodbye.
ftp>

Transferring Files:

Typically files may be transferred from the local system to the remote system or from the remote system to the local system. Some systems only allow files to be copied from them; for security reasons they do not allow files to be written to them.

There are two commands to transfer files: **get** and **put**. Both of these commands are made in reference to the local system. Therefore **get** copies a file from the remote system to the local system, whereas **put** writes a file from the local system to the remote system.

When we ftp a file, we must either be in the correct directories or use a file path to locate and place the file. The directory can be changed on the remote system by using the change directory (**cd**) command within ftp.

There are several commands that let us change the remote file directory. For example we can change a directory, create a directory, and remove a directory. We can also list the remote directory. These commands work just like their counterparts in UNIX. A complete list of ftp commands is shown in table given below:

!	debug	Mdir	pwd	Size
$	dir	Mget	quit	Status
account	direct	Mkdir	quote	struct
append	disconnect	Mls	recv	sunique
ascii	form	Mode	reget	system
bell	get	Modtime	rename	tenex
binary	glob	Mput	reset	trace
bye	hash	Newer	restart	type
case	help	Nlist	rhelp	umask
cd	idle	Nmap	rmdir	user
cdup	image	ntrans	rstatus	verbose
chmod	lcd	open	ruinque	win
close	ls	prompt	send	?
cr	macdef	proxy	sendport	
Delete	mdelete	put	site	

vi EDITOR:

The **vi** editor is the interactive part of **vi/ex**. When initially entered, the text fills the buffer, and one screen is displayed. If the file is not large enough to fill the screen, the empty lines below text on the screen will be identified with a tilde (~) at the beginning of each line. The last line on the file is a **status line;** the status line is also used to enter **ex** commands.

Commands:

Commands are the basic editing tools in **vi.** As a general rule, commands are case sensitive. This means that a lowercase command and its corresponding uppercase command are different, although usually related.

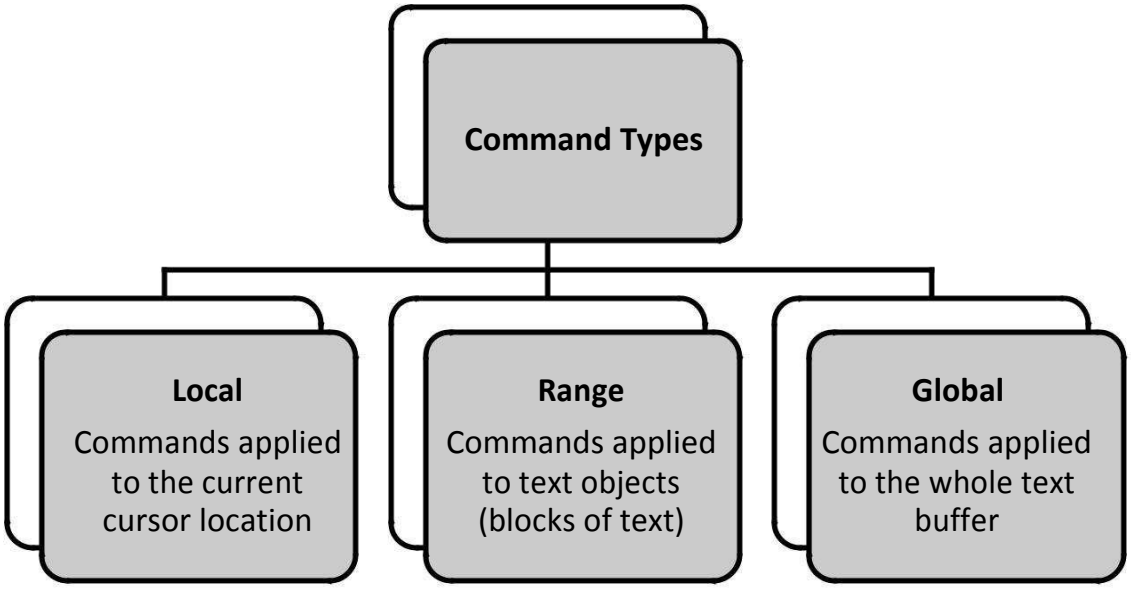
For example the lower case insert command (i) inserts before the cursor, whereas the uppercase insert command (I) inserts at the beginning of the line.

One of the things that most bothers new vi users is that the command is not seen. When we key the insert command, we are automatically in the insert mode, but there is no indication on the screen that anything has changed.

Another problem for new users is that commands do not require a Return key to operate. Generally command keys are what are known as hot keys, which means that they are effective as soon as they are pressed.

COMMAND CATEGORIES:

The commands in vi are divided in to three separate categories: local commands, range commands, and global commands as shown in figure below. **Local commands** are applied to the text at the current cursor. The **range commands** are applied to blocks of text are known as text objects. **Global commands** are applied to all of the text in the whole buffer, as contrasted to the current window.



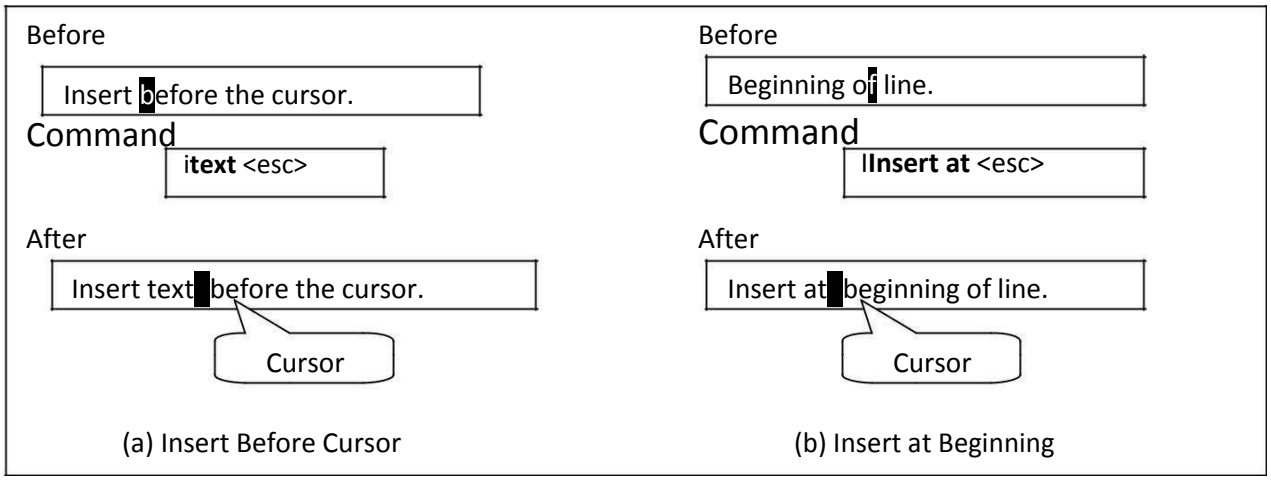
LOCAL COMMANDS IN vi

Local commands are commands are applied to the text relative to the cursors current position. We call the character at the cursor the current character, and the line that contains the cursor is the current line. The local commands are discussed below:

Insert Text Commands (i, I)

We can insert text before the current position or at the beginning of the current line. The lowercase insert command (i) changes to the text mode. We can then enter text. The character at the cursor is pushed down the line as the new text is inserted. When we are through entering text, we return to the command mode by keying *esc*.

The uppercase insert (I) opens the beginning of the current line for inserting text. The following figure shows the examples of character insertion.



(a) Insert Before Cursor (b) Insert at Beginning

Append Text Commands (a, A)

The basic concept behind all append commands is to add the text after a specified location. In vi, we can append after the current character (a) or after the current line (A). The following figure demonstrates the append command.

Newline Commands (o, O)

The newline command creates an open line in the text file and inserts the text provided with the command. Like insert and append, after entering the text, we must use the escape key to return to the command mode. To add the text in a new line below the current line, use the lowercase newline command (o). To add the new text in a line above the current line, use the uppercase newline command (O). The following figure demonstrates the newline command.

Replace Text Commands (r, R)

When we replace text, we can only change text in the document. We cannot insert the text nor can we delete old text. The lowercase replace command (r) replaces a single character with another character and immediately returns to command mode. It is not necessary to use the Escape key. With the uppercase replace command (R), on the other hand, we can replace as many characters as necessary. Every keystroke will replace one character in the file. To return to the command mode in this case, we need to use the escape key. The following figure demonstrates the replace command.

Note: At the end of both the commands the cursor is on the last character replaced.

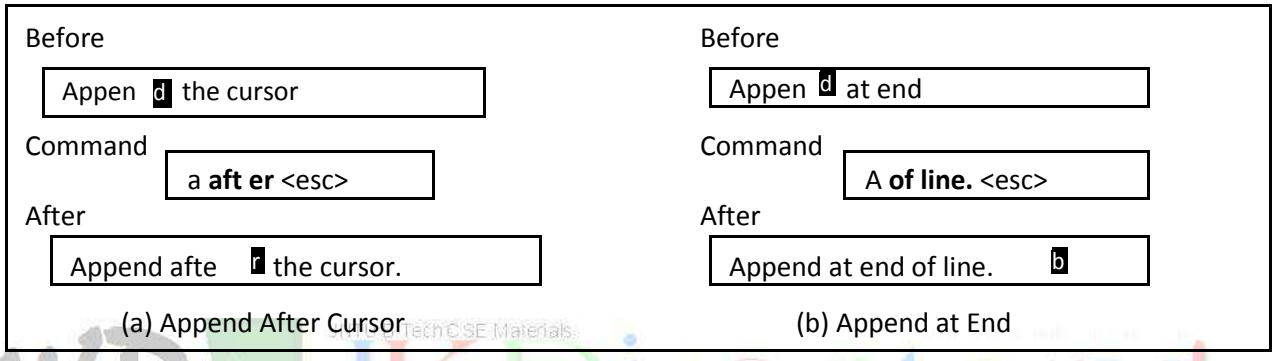


Figure: Append text after current character/line

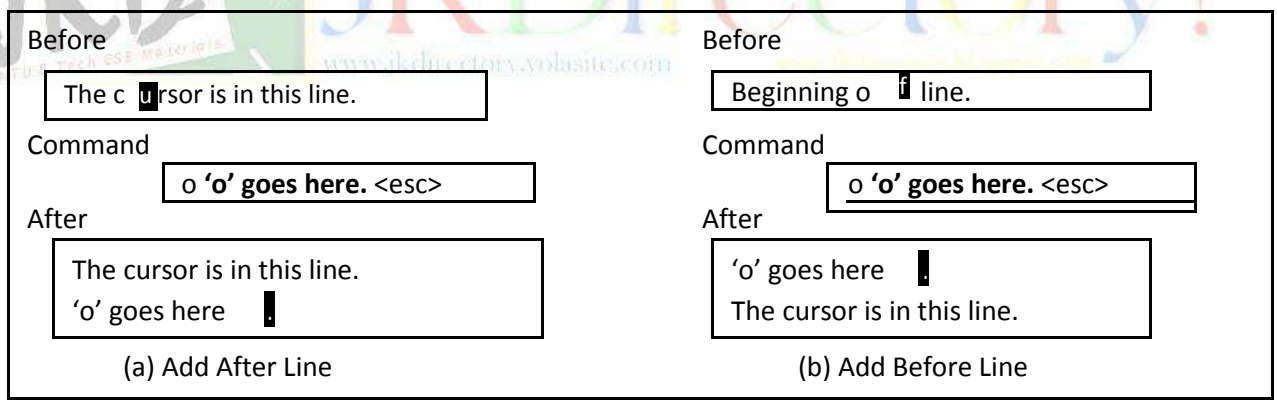


Figure: Newline Command

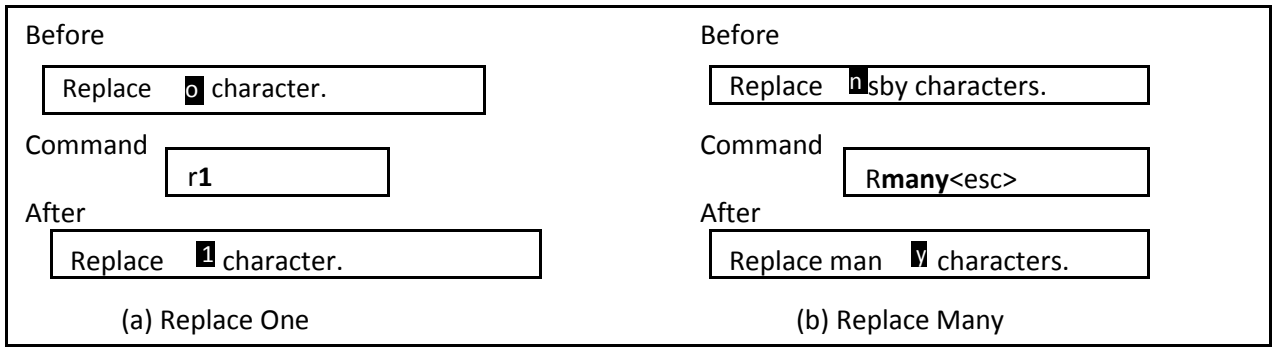


Figure: Replace Commands

Substitute Text Commands (s, S)

Whereas the replace text command is limited to the current characters in the file, the lowercase substitute (s) replaces one character with one or more characters. It is especially useful to correct spelling errors. The uppercase substitute command (S) replaces the entire current line with new text. Both substitute commands require an Escape to return to the command mode.

Delete Character Commands (x, X)

The delete character command deletes the current character or the character before the current character. To delete the current character, use the lowercase delete command (x). To delete the character before the cursor, use the uppercase delete command (X). After the character has been deleted, we are still in the command mode, so no Escape is necessary.

Mark Text Command (m)

Marking text places an electronic "finger" in the text so that we can return to it whenever we need to. It is used primarily to mark a position to be used later with a range command. The mark command (m) requires a letter that becomes the name of the marked location.

The letter is invisible and can be seen only by vi. Marked locations in the file are erased when the file is closed; they are valid only for the current session and are not permanent locations.

Change Case Command (~)

The change case command is the tilde (~). It changes the current character from upper- to lowercase or from lower-to uppercase. Only one character can be changed at a time, but the cursor location is advanced so that multiple characters can easily be changed. It can also be used with the repeat modifier to change the case of multiple characters. Nonalphabetic characters are unaffected by the command.

Put Command (p, P)

In a word processor, the put command would be called "paste". As with many other commands, there are two versions, a lowercase p and an uppercase P. The lowercase put copies the contents of the temporary buffer after the cursor position. The uppercase put copies the buffer before the cursor. The exact placement of the text depends on the type of data in the buffer. If the buffer contains a character or a word, the buffer contents are placed before or after the cursor in the current line.

If the buffer contains a line, sentence, or paragraph, it is placed on the previous line or the next line.

Join Command (J)

Two lines can be combined using the join command (J). The command can be used anywhere in the first line. After the two lines have been joined, the cursor will be at the end of the first line.

RANGE COMMANDS IN vi

A range command operates on a text object. The vi editor consider the whole buffer as a long stream of characters of which only a portion may be visible in the screen window. Because range commands can have targets that are above or below the current window, they can affect unseen text and must be used with great care.

There are four range commands in vi: move cursor, delete, change, and yank. The object of each command is a text object and defines the scope of the command. Because the range commands operate on text objects, we begin our discussion of range commands with text objects.

Text Object:

A text object is a section of text between the two points: the cursor and a target. The object can extend from the cursor back toward the beginning of the document or forward toward the end of the document. The target is a designator that identifies the end of the object being defined.

The object definitions follow a general pattern. When the object is defined before the cursor, it starts with the character to the left of the cursor and extends to the beginning of the object. When the object is defined after the cursor, it begins with the cursor character and extends to the end of the object.

Object ranges:

To left:	Starts with character on left of cursor to the beginning of the range.
To right:	Starts with cursor character to the end of range.

There are seven general classes of objects: character, word, sentence, line, paragraph, block, and file. All text objects start with the cursor.

Character Object: A character object consists of only one character. This means that the beginning and end of the object are the same. To target the character immediately before the cursor, we use the designator h. To target the current character, we use the designator l.

Word Object: A word is defined as a series of non whitespace characters terminated by character. A word object may be a whole word or a part of a word depending on the current character location (cursor) and the object designator. There are three word designators.

The designator b means to go backward to the beginning of the current word; if the cursor is at the beginning of the current word, it moves to the beginning of the previous word.

The designator w means to go forward to the beginning of the next word, including the space between the words if any.

The designator e means to go forward to the end of the current word; it does not include the space.

Line Object: A line is all of the text beginning with the first character after a newline to the next newline. A line object may consist of one or more lines. There are six line designators:

To target the beginning of the current line, use 0 (zero)

To target the end of the line, use $

To target the beginning of the line above the current line, use – (minus)

To target the beginning of the next line, user +.

To target the character immediately above the current character, use k.

To target the character immediately below the current character, use j.

Sentence Object: A sentence is a range of text that ends in a period or a question mark followed by two spaces or a new line. This means that there can be many lines in one sentence. A sentence designator defines a sentence or a part of a sentence as an object. There are two sentence designators 1) **(** and 2) **)**. To remember the sentence targets, just think of a sentence enclosed in parenthesis.

The open parenthesis is found at the beginning of the sentence; it selects the text from the character immediately before the cursor to the beginning of the sentence. The close parenthesis selects the text beginning from and including the cursor to the end of the sentence, which includes the two spaces that delimit it. Both sentence objects work consistently with all operations.

Paragraph Object: A paragraph is a range of text starting with the first character in the file buffer or the first character after a blank line (a line consisting only of a new line) to the next blank line or end to the buffer. Paragraph may contain one or more sentences. There are two paragraph designators 1) **}** and 2) **{**. To remember the paragraph targets, think of them as enclosed in braces, which are bigger than parenthesis.

The left brace targets the beginning of the paragraph including the blank line above it. The right brace targets the end of the paragraph but does not include the blank line.

Block Object: A block is a range of text identified by a marker. It is the largest range object and can span multiple sentences and paragraphs. In fact, it can be used to span the entire file. The two block designators are `a and ‘a.

Screen Objects: We can define the part of the text or the whole screen as a screen object. There are three simple screen cursor moves:

H: Moves the cursor to the beginning of the text line at the top of the screen.

L: Moves the cursor to the beginning of the text line at the bottom of the screen.

M: Moves the cursor to the beginning of the line at the middle of the screen.

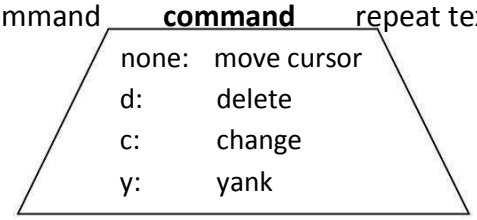
File Object: There is one more target, the file. To move to the beginning of the last line in the file object, or to create a range using it, use the designator G.

Text Object Commands:

Four vi commands are used with the text objects: move cursor, delete, change and yank.

The range commands share a common syntactical format that includes the command and three modifiers. The format is shown below:

repeat command **command** repeat text object text object



Yank command: The yank command copies the current text object and places it in a temporary buffer. It parallels the delete command except that the text is left in the current position. The typical use of the yank command is the first step in a copy and paste operation. After the text has been placed in the temporary buffer, we move to an insert location and then use the put command to copy the text from the buffer to the file buffer.

Global Commands in vi:

Global commands are applied to the edit buffer without reference to the current position.

Scroll Commands:

Scroll commands affect the part of the edit buffer (file) that is currently in the window. They are summarized in the table given below:

Command	Function
ctrl + y	Scrolls up one line.
ctrl + e	Scrolls down one line.
ctrl + u	Scrolls up half a screen.
ctrl + d	Scroll down half a screen.
ctrl + b	Scrolls up whole screen.
ctrl + f	Scrolls down whole screen.

Undo Commands:

The vi editor provides a limited undo facility consisting of two undo commands: undo the most recent change (u) and restore all changes to the current line (U).

Repeat Command:

We can use the dot command (.) to repeat the previous command. For example, instead of using the delete line (dd) several times, we can use it once and then use the dot command to delete the remaining lines. The dot command can be used with all three types of vi commands: local, range and global.

Screen Regeneration Commands:

There are times when the screen buffer can become cluttered and unreadable. For example, if in the middle of an edit session someone sends you a message; the message replaces some of the text on the screen but not in the file buffer. At other times, we simply want to reposition the text. Four vi commands are used to regenerate or refresh the screen. These four commands are summarized in the table given below:

Command	Function
z return	Regenerates screen and positions current line at top.
z.	Regenerates screen and positions current line at middle.
z-	Regenerates screen and positions current line at bottom.
ctrl + L	Regenerates screen without moving cursor.

Display Document Status Line:

To display the current document status in the status line at the bottom of the screen, we use the status command (ctrl + G). The status fields across the line are the filename, a flag indicating if the file has been modified or not, the position of the cursor in the file buffer, and the current line position as a percentage of the lines in the file buffer.

For example, if we are at line 36 in our file, *TheRaven*, we would see the following status when we enter the ctrl + G:

"TheRaven" [Modified] line 36 of 109 - - 33%- -

When we are done with the edit session, we quit vi. To quit and save the file, we use the vi zz command. If we don't want to save the file, we must use ex's quit and don't save command, q!

Rearrange Text in vi:

Most word processors have cut, copy, and paste commands. vi's equivalents use delete and yank commands to insert the text in to a buffer and then a put command to paste it back to the file buffer. In this section we will see how to use these commands to move and copy text.

Move Text:

When we move the text, it is deleted from its original location in the file buffer and then put in to its new location. This is the classic cut and paste operation in a word processor. There are three steps required to move text:

1. Delete text using the appropriate delete command, such as delete paragraph.
2. Move the cursor to the position where the text is to be displayed.
3. Use the appropriate put command (p or P) to copy the text from the buffer to the file buffer.

In the following example, we move three lines starting with the current line to the end of the file buffer. To delete the lines, we use the delete line command with a repeat object modifier. We then position the cursor at the end of the file buffer and copy the text from the buffer.

```
3dd     # Delete three lines to buffer

G       # Move to end of file buffer

p       # put after current line
```

Copy Text:

When we copy text, the original text is left in the file buffer and also put (pasted) in a different location in the buffer. There are two ways to copy text. The preferred method is to yank the text. Yank leaves the original text in place. We then move the cursor to the copy location and put the text. The steps are as follows:

1. Yank text using yank block, y.
2. Move cursor to copy location.
3. Put the text using p or P.

The second method is to delete the text to be copied, followed immediately with a put. Because the put does not empty the buffer, the text is still available. Then move to the copy location and put again. The following code yanks a text block and then puts it in a new location:

Move cursor to the beginning of the block

ma # set mark 'a'

manually move cursor to the end of the block y`a # yank block 'a' leaving it in place

manually move cursor to copy location p # put text in new location

Named Buffers:

The vi editor uses one temporary buffer and 35 named buffers. The text in all buffers can be retrieved using the put command. However, the data in the temporary buffer are lost whenever a command other than a position command is used. This means that a delete followed by an insert empties the temporary buffer.

Then named buffers are available until their contents are replaced. The first time named buffers are known as the numeric buffers because they are identified by the digits 1 through 9. The remaining 26 named buffers are known as the alphabetic buffers; they are identified by the lower case letters a through z.

To retrieve the data in the temporary buffer, we use the basic put command. To retrieve the text in a named buffer, we preface the put command with a double quote, and the buffer name as shown in the following command syntax. ***Note that there is no space between the double quote, the buffer name, and the put command.***

"buffer-namep

Using this syntax, the following three examples would retrieve the text in the temporary buffer, buffer 5 and buffer k.

P "5p "kP

Numeric Named Buffers:

The numeric named buffers are used automatically whenever sentence, line, paragraph, screen, or file text is deleted. The deleted text is automatically copied to the first buffer (1) and is available for later reference. The deleted text for the previous delete can be retrieved by using either the put command or by retrieving numeric buffer 1.

Each delete is automatically copied to buffer 1. Before it is copied, buffer 8 is copied to buffer 9, buffer 7 is copied to buffer 8, and so forth until buffer 1 has been copied to buffer 2. The current delete is then placed in buffer 1. This means that at any time the last nine deletes are available for retrieval.

If a repeat modifier is used, then all of the text for the delete is placed in the buffer 1. In the following example, the next three lines are considered as one delete and are placed in buffer 1: 3dd

Alphabetic Named Buffers:

The alphabetic named buffers are to save up to 26 text entities. Whereas the numeric buffers can only store text objects that are at least a sentence long, we can store any size object in an alphabetic buffer. Any of the delete or yank commands can be used to copy data to an alphabetic buffer.

To use the alphabetic buffers, we must specify the buffer name in the delete or yank command. As previously stated, the buffer names are the alphabetic characters a through z. The buffer name is specified with a double quote followed immediately by the buffer name and the yank command as shown below:

"ad # Delete and store line in 'a' buffer
"my # Yank and store line in 'm' buffer

Once the text has been stored, it is retrieved in the same way we retrieved data from the numeric buffers. To retrieve the two text objects created in the previous example, we would enter:

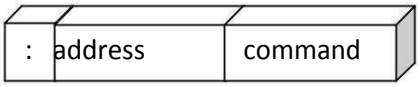
"mp # Retrieve line in 'm' buffer after current cursor
"ap # Retrieve line in 'a' buffer before current cursor

Overview of ex Editor:

The ex editor is a line editor. When we start ex, we are given its colon prompt at the bottom of the screen. Each ex instruction operates on one specific line. ex is the integral part of the vi editor. We can start in either of these editors, and from each, we can move to the other.

ex Instruction Format:

An ex instruction is identified by its token, the colon (:). Following the token is an address and a command. The ex instruction format is shown in figure below:



Addresses:

Every line in the file has a specific address starting with one for the first line in the file. An ex address can be either a single line or a range of lines.

Single Line:

A single line address defines the address for one line. The five single line address formats are discussed below:

Current line: The current line, represented by a period (.), is the last line processed by the previous operation. When the file is opened, the last line read into the buffer (i.e. the last line in the file) is the current line. In addition to the period, if no address is specified, the current line is used.

Top of Buffer: The top of the buffer, represented by zero (0), is a special line designation that sets the current line to the line before the first line in the buffer. It is used to insert lines before the first line in the buffer.

Last Line: The last line of the buffer is represented by the dollar sign ($).

Line Number: Any number within the line range of the buffer (1 to $) is the address of that line.

Set-of-Line Addresses:

There are two pattern formats that can be used to search for a single line. If the pattern search is to start with the current line and search forward in the buffer – that is, toward the end of the buffer – the pattern is enclosed in slashes (/ . . . /). If the search is to start with the current line and move backward – that is, toward the beginning of the buffer (1) – the pattern is enclosed in question marks (?).

Note however in both searches, if the pattern is not found, the search wraps around to completely search the buffer. If we start forward search at line 5, and the only matching line is in line 4, the search will proceed from line 5 through line $ and then restart at line 1 and search until it finds the match at line 4.

The following table summarizes these two addresses:

Address	Search Direction
/pattern/	Forward
?pattern?	Backward

Range Addresses:

A range address is used to define a block of consecutive lines. If has three formats as shown in the table given below:

Address	Range
%	Whole file
address1, address2	From address1 to address2 (inclusive)
address1;address2	From address1 to address2 relative to address1 (inclusive)

The last two addresses are similar; the only difference is one uses a comma separator and one uses a semicolon. To see the differences, therefore, we need to understand the syntactical meaning of the comma and the semicolon separators.

Comma Separator: When the comma is used, the current address is unchanged until the complete instruction has been executed. Any address relative to the current line is therefore relative to the current line when the instruction is given.

Semicolon Separator: When we use a semicolon, the first address is determined before the second address is calculated. It sets the current address to the address determined by the first address.

Commands: There are many ex commands. Some of the basic commands that are commonly used are given in the following table:

Command	Description
d	Delete
co	Copy
m	Move
r	Read
w	Write to file
y	Yank

Command	Description
P	Print (display)
Pu	Put
Vi	Move to vi
w filename	Write file and continue editing
S	Substitute
Q	Quit
q!	Quit and do not save
Wq	Write and quit
X	Write file and exit

Delete Command (d): The delete command (d) can be used to delete a line or range of lines. The following example deletes lines 13 through 15:

:13,15d

Copy Command (co): The copy command (co) can be used to copy a single line or a range of lines after a specified line in the file. The original text is left in place. The following examples copy lines 10 through 20 to the beginning of the file (0), the end of the file ($), and after line 50:

:10,20co 0
:10,20co $
:10,20c0 50

Move Command (m): The format of the move command (m) is the same as the copy command. In a move, however, the original text is deleted.

Read and Write Commands (r, w): The read command (r) transfers lines from a file to the editor's buffer. Lines are read after a single line or a set of lines identified by a pattern. The write command (w file_name), on the other hand, can write the whole buffer or a range within the buffer to a file. All address ranges (except nested range) are valid.

$ **ex file1**
"file1" 5 lines, 10 characters
:**r file2**
"file2" 6 lines, 330 characters
:**w TempFile**
"TempFile" [New file] 11 lines, 340 characters
:**q**
$ **ls –l TempFile**
-rw-r- -r- - 1 gilberg staff 340 Oct 18 18:16 TempFile

Print Command (p): The print command (p) displays the current line or a range of lines on the monitor.

Move to vi Command (vi): The move to vi command (vi) switches the editor to vi. The cursor is placed at the current line, which will be at the top of the monitor. Once in vi, however, you will remain in it as though you had started it. You will be able to execute ex command, but after each command, you will automatically return to vi.

Substitute Command (s): The Substitute command (s) allows us to modify a part of line or a range of lines. Using substitute, we can add text to a line, delete text, or change text. The format of the substitute command is:

addresss/pattern/replacement-string/flag

Quit Command (q, q!, wq): The ex quit command (q) exits the editor and returns to the UNIX command line. If the file has been changed, however, it presents an error message and returns to the ex prompt. At this point we have two choices. We can write the file and quit (wq), or we can tell ex to discard the changes (q!).

Exit Command (x): The exit command (x) also terminates the editor. If the file has been modified, it is automatically written and the editor terminated. If for any reason, such as write permission is not set, the file cannot be written, the exit fails and the ex prompt is displayed.

ATOMS AND OPERATORS:

A regular expression is a pattern consisting of a sequence of characters i.e. matched against text. A regular expression is like a mathematical expression. A mathematical expression is made of operands (data) and operators. Similarly, a regular expression is made of atoms and operators.

The **atom** specifies what we are looking for and where in the text the match is to be made. The **operator,** which is not required in all expressions, combines atoms into complex expressions.

ATOMS: An atom in a regular expression can be one of the five types: a single character, a dot, a class, an anchor, or a back reference.

Single Character: The simplest atom is a single character. When a single character appears in a regular expression, it matches itself. In other words, if a regular expression is made of one single character, that character must be somewhere in the text to make the pattern match successful.

Dot: A dot matches any single character except the newline character (\n). This universal matching capability makes it a very powerful element in the operation of regular expressions. By itself, however, it can do nothing because if matches everything. Its power in regular expressions comes from its ability to work with other atoms to create an expression.

For example consider the following example:

a.

This expression combines the single-character atom, a, with the dot atom. It matches any pair of characters where the first character is a. Therefore, it matches aa, ah, ab, ax, and a5, but it does not match Aa.

Class: The class atom defines a set of ASCII characters, any one of which may match any of the characters in the text. The character set to be used in the matching process is enclosed in brackets. The class set is a very powerful expression component. Its power is extended with three additional tokens: ranges, exclusion, and escape characters. A range of text characters is indicated by a dash (-). Thus the expression [a-d] indicates that the characters a and b and c and d all included in the set.

Sometimes it is easier to specify which characters are to be excluded from the set – that is to specify its complement. This can be done using exclusion, which is the UNIX *not* operator (^). For example to specify any character other than a vowel, we would use [^aeiou].

The third additional token is the escape character (\). It is used when the matching character is one of the other two tokens. For example to match a vowel or a dash, we would use the escape character to indicate that the dash is a character and not a range token. This example is coded as [aeiou\-].

Anchors: Anchors are atoms that are used to line up the pattern with a particular part of a string. In other words anchors are not matched to the text, but define where the next character in the pattern must be located in the text. There are four types of anchors: beginning of line (^), end of line ($), beginning of word (\<), and end of word (\>).

Anchors are another atom that is often used in combinations. For example, to locate the string that begins with letter Q, we would use the expression ^Q. similarly to find a word that ends in g, we would code the expression as g\>.

Back references: We can temporarily save text in one of the nine save buffers. When we do we refer the text in a saved buffer using a back reference. A back reference is coded using the escape character and a digit in the range of 1 to 9 as shown below:

\1 \2 . . . \9

A back reference is used to match text in the current or designated buffer with text that has been saved in one of the systems nine buffers.

OPERATORS:

To make the regular expressions more powerful, we can combine atoms with operators. The regular expression operators play the same role as mathematical operators. Mathematical expression operators combine mathematical atoms (data); regular expression operators combine regular expression atoms.

We can group the regular expressions in to five different categories: *sequence operators, alternation operators, repetition operators, group operators,* and *save operators.*

The **sequence** operator is *nothing*. This means that if a series of atoms such as a series of characters are shown in a regular expression, it is implied that there is an invisible sequence operator between them. Examples of sequence operators are shown below:

Dog	➔	matches the pattern “Dog”
a . . b	➔	matches “a”, any two characters, and “b”
[2 – 4] [0 – 9]	➔	matches a number between 20 and 49
[0 – 9] [0 – 9]	➔	matches any two digits
`$	➔	matches a blank line
^.$	➔	matches a one-character line
[0 – 9] – [0 – 9]	➔	matches two digits separated by a “–“

The **Alternation** operator (|) is used to define one or more alternatives. For example if we want to select between A or B, we would code the regular expression as A | B. Alternation can be used with single atoms, but it is usually used for selecting between two or more sequences of characters or groups of characters. That is, the atoms are usually sequences.

For single alternation we suggest that you use the class operator. An example of alternation among sequences is presented in example below:

UNIX | unix	➔	Matches “UNIX” or “unix”
Ms|Miss|Mrs	➔	Matches “Ms” or “Miss” or “Mrs”

The **repetition** operator is a set of escaped braces (\ { … \ }) that contains two numbers separated by a comma. It specifies that the atom or expression immediately before the repetition may be repeated. The first number (m) indicates the minimum required times the previous atom must appear in the text.

The second number (n) indicates the maximum number of times it may appear. For example \ { 2, 5 \ } indicates that the previous atom may be repeated two to five times.

Example:

A\ {3, 5\} ➔ matches “AAA”, “AAAA”, or “AAAAA”
BA\ {3, 5\} ➔ matches “BAAA”, “BAAAA”, or “BAAAAA”

Basic Repetition Forms:

The m and n values are optional, although at least one must be present. That is either may appear without the other. If only one repetition value (m) is enclosed in the braces, the previous atom must be repeated exactly m times – no more, no less. E.g. \ {3 \}.

If the minimum value (m) is followed by a comma without a maximum value, the previous atom must be present at least m times, but it may appear more than m times. In the following example the previous atom may be repeated three or more times but no less than three times. E.g. \ {3, \}.

If the maximum value (n) is preceded by a comma without a minimum, the previous atom may appear up to n times and no more. In the following example the previous atom may appear zero to three times, but no more. E.g. \ {, 3\}

Short Form Operators:

Three forms of repetition are so common that UNIX has special shortcut operators for them. The asterisk (*) may be used to repeat an atom zero or more times. (It is same as \ {0 , \}). The plus (+) is used to specify that the atom must appear one or more times. (It is same as \ {1, \}). The question mark (?) is used to repeat the pattern zero or one time only. (It is same as \ {0, 1\}).

The **group** operator is a pair of opening and closing parentheses. When a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only to the previous character. In the following example, the group (BC) must be repeated exactly three times.

A(BC)\ {3 \} ➔ matches ABCBCBC

The **save** operator which is a set of escaped parentheses, \ (… \), copies a matched text string to one of the nine buffers for later reference. Within an expression, the first saved text is copied to buffer 1, the second saved text is copied to buffer 2 and so forth for up to nine buffers. Once text has been saved, it can be referred to by using a back reference.

GREP FAMILY AND OPERATIONS:

The command **grep** stands for **g**lobal **r**egular **e**xpression **p**rint. It is the family of programs that is used to search the input file for all lines that match a specified regular expression and write them to the standard output file (monitor). The format of grep is shown below:

grep ***options*** **regexp** **filelist**

Options:

-b: print block numbers -c: print only match count -i: ignore upper-/lowercase

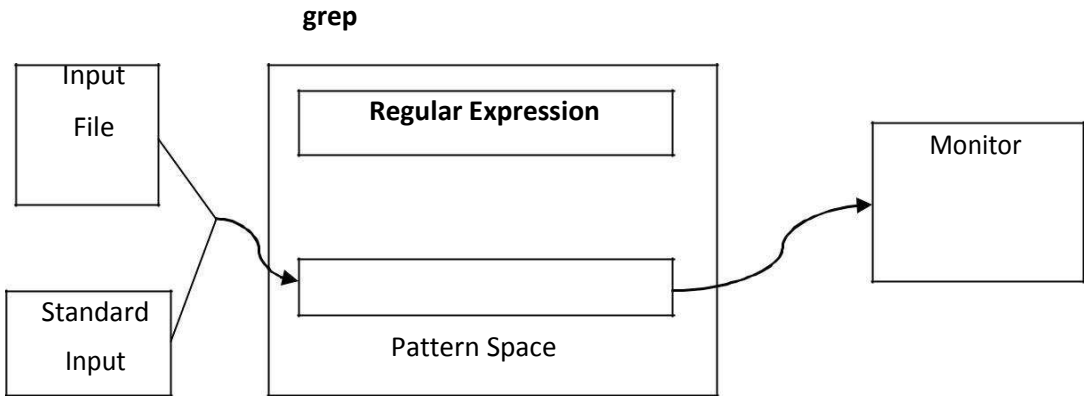
-l: print files with at least one match -n: print line numbers

-s: silent mode; no output

-v: print lines that do not match -x: print only lines that match -f file: expressions are in file

OPERATION:

To write scripts that operate correctly, you must understand how the grep utilities work. We begin, therefore with a short explanation of how they work.



For each line in the standard input (input file or keyboard), grep performs the following operations:

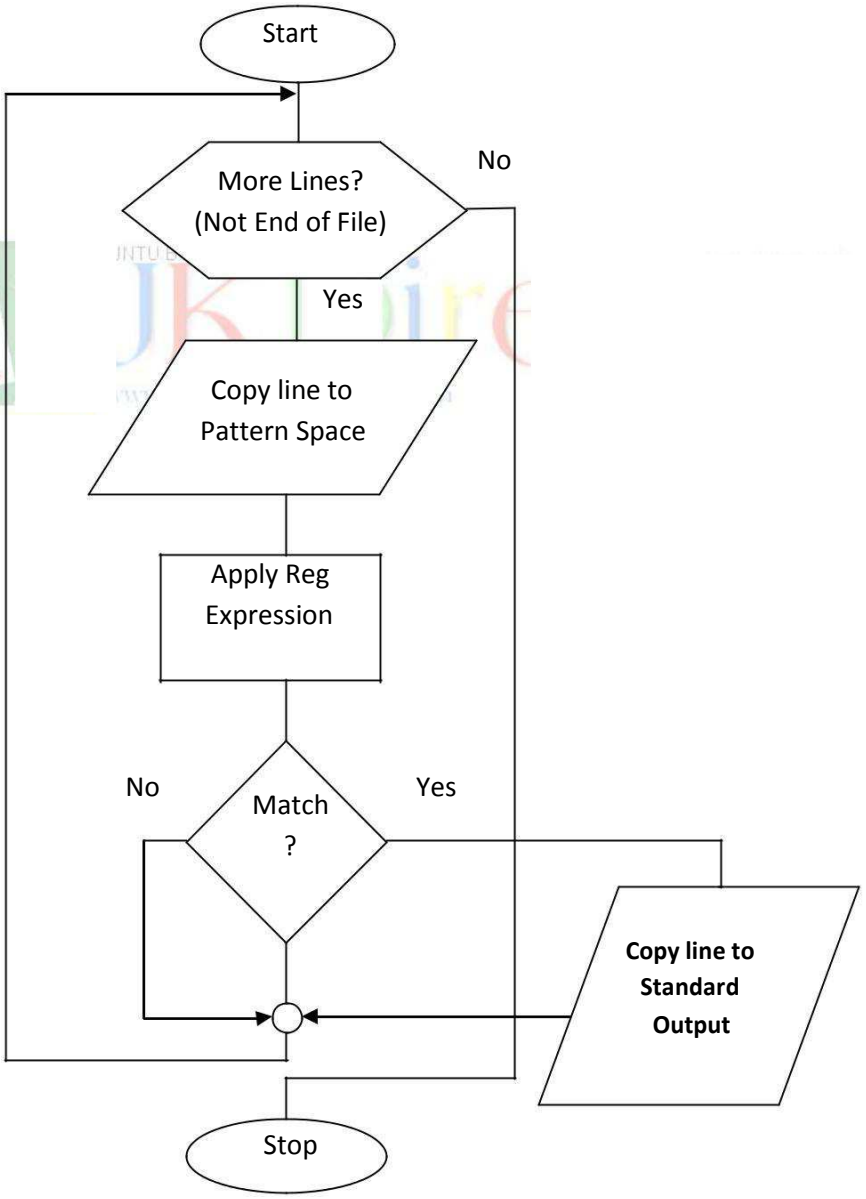
1. Copies the next input line into the pattern space. The pattern space is a buffer that can hold only one text line.

2. Applies the regular expression to the pattern space.

3. If there is a match, the line is copied from the pattern space to the standard output. The grep utilities repeat these three operations on each line in the input.

grep flowchart:

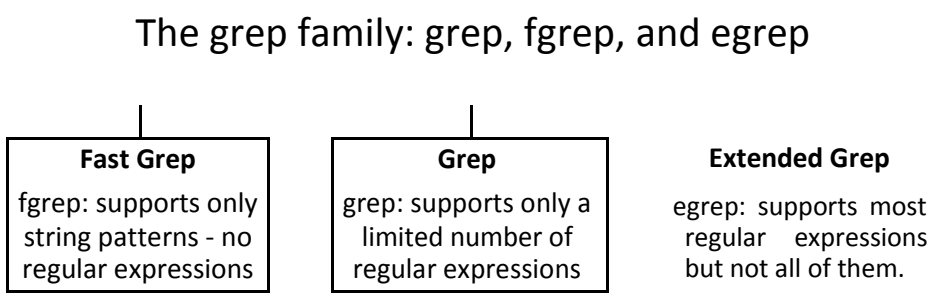
Another way to look at how grep works is to study the flowchart of its operations. Two points about the grep flowchart in the figure below need to be noted. First, the flowchart assumes that no options were specified. Selecting one or more options will change the flowchart. Second although grep keeps a current line counter so that it always knows which line is being processed, the current line number is not reflected in the flowchart.



grep Family:

There are three utilities in the grep family: grep, egrep, and fgrep. All these search one or more files and output lines that contain the text that matches criteria specified as a regular expression. The whole line does not have to match the criteria; any matching text in the line is sufficient for it to be output.

It examines each line in the file, one by one. When a line contains matching pattern, the line is output. Although this is a powerful capability that quickly reduces a large amount of data to a meaningful set of information, it cannot be used to process only a portion of the data. The grep family appears in the figure given below:



grep Family Options:

There are several options available to the grep family. A summary is found in the table given below:

Option	Explanation
-b	Precedes each line by the file block number in which it is found.
-c	Prints only a count of the number of lines matching the pattern.
-i	Ignores upper-/lowercase in matching text.
-l	Prints a list of files that contain at least one line matching the pattern.
-n	Shows line number of each line before the line.
-s	Silent mode. Executes utility but suppresses all output.
-v	Inverse output. Prints lines that do not match pattern.
-x	Prints only lines that entirely match pattern.
-f file	List of strings to be matched are in file.

grep Family Expressions:

The fast grep (fgrep) uses only sequence operators in a pattern; it does not support any of the other regular expression operators. Basic **grep** and extended grep (**egrep**) both accept regular expressions. As you can see from the table below not all expressions are available.

Atoms	grep	fgrep	egrep	Operators	grep	fgrep	egrep
Character	√	√	√	Sequence	√	√	√
Dot	√		√	Repetition	all but ?		* ? +
Class	√		√	Alternation			√
Anchors	√		^ $	Group			√
Back Reference	√			Save	√		

Expressions in the **grep** utilities can become quite complex, often combining several atoms and/or operators into one large expression. When operators and atoms are combined, they are generally enclosed in either single quotes or double quotes. Technically the quotes are needed only when there is blank or other character that has a special meaning to the **grep** utilities. The combined expression format is shown below:

$**grep 'Forouzan, *Behrouz' file1**

grep:

The original of file matching utilities, grep handles most of the regular expressions. grep allows regular expressions but is generally slower than egrep. Use it unless you need to group expressions or use repetition to match one or more occurrences of a pattern. It is the only member of the grep family that allows saving the results of a match for later use.

In the following example we use grep to find all the lines that end in a semicolon (;) and then pipe the results to **head** and print the first two.

$ grep –n ";$" TheRaven | head -2

8:Ah, distinctly I remember it was in the bleak December;
16:Thrilled me - - filled me with fantastic terrors never felt before;

From the table above we see the –n option requests that the line numbers from the original file be included in the output. They are seen at the beginning of each line. The regular expression **;$** looks for a semicolon (;) at the end of the line ($). The file name is "TheRaven", and the output of the **grep** execution is piped to the head utility where only the first two lines are printed (-2).

Fast grep:

If your search criteria require only sequence expressions, fast grep (**fgrep**) is the best utility. Because its expressions consist of only sequence operators, it is also easiest to use if you are searching for text characters that are the same as regular expression operators such as the escape, parentheses, or quotes. For example, to extract all lines of the file that contain an apostrophe, we could use **fgrep** as $ **fgrep –n " ' " file name**

Extended grep:

Extended grep (egrep) is the most powerful of the three grep utilities. While it doesn't have the save option, it does allow more complex patterns. Consider the case where we want to extract all lines that start with a capital letter and end in an exclamation point (!). Our first attempt at this command is shown as follows: $ **egrep –n '^[A-Z].*!$' filename**

The first expression starts at the beginning of the line (^) and looks at the first character only. It uses a set that consists of only uppercase letters ([A-Z]). If the first character does not match the set, the line is skipped and the next line is examined.

If the first character is a match, the second expression (.*) matches the rest of the line until the last character, which must be an exclamation mark; the third expression examines the character at the end of the line ($). It must be an explanation point (a bang). The complete expression therefore matches any line starting with an uppercase letter, that is followed by zero or more characters, and that ends in a bang.

Finally note that we have coded the entire expression in a set of single quotes even though this expression does not require it.

Examples:

1. Count the number of blank lines in the file → $ **egrep –c '^$' testFile**
2. Count the number of nonblank lines in the file → $ **egrep –c '.' testFile**
3. Select the lines from the file that have the string UNIX → $ **fgrep 'UNIX' testFile**
4. Select the lines from the file that have only the string UNIX → $ **egrep '^UNIX$' testFile**
5. Select the lines from the file that have the pattern UNIX at least two times. $ **egrep 'UNIX.*UNIX' testFile**

SEARCHING FOR FILE CONTENTS:

Some modern operating systems allow us to search for a file based on a phrase contained in it. This is especially handy when we have forgotten the filename but know that it contains a specific expression or set of words. Although UNIX doesn't have this capability, we can use the grep family to accomplish the same thing.

Search a Specific Directory:

When we know the directory that contains the file, we can simply use grep by itself. For example, to find a list of all files in the current directory that contain "Raven," we would use the search in the example given below:

$ **ls**

RavenII TheRaven man.ed regexp.dat $ **grep –l 'Raven'**
*****

RavenII

TheRaven

The option l prints out the filename of any file that has at least one line that matches the grep expression.

Search All Directories in a Path:

When we don't know where the file is located, we must use the **find** command with the execute criterion. The **find** command begins by executing the specified command, in this case a grep search, using each file in the current directory. It then moves through the subdirectories of the current file applying the grep command. After each directory, it processes its subdirectories until all directories have been processed. In the example below, we start with our home directory (~).

$ **find ~ –type f –exec grep –l "Raven" {} \;**

Overview of sed and awk:

Sed is an acronym for **s**tream **ed**itor. Although the name implies editing, it is not a true editor; it does not change anything in the original file. Rather sed scans the input file, line by line, and applies a list of instructions (called a sed script) to each line in the input file. The script, which is usually a separate file, can be included in the sed command line if it is a one-line command. The format is shown below:

sed options script file list

The sed utility has three useful options. Option –n suppresses the automatic output. It allows us to write scripts in which we control the printing. Option –f indicates that there is a script file, which immediately follows on the command line. The third option –e is the default. It indicates that the script is on the command line, not in a file. Because it is the default, it is not required.

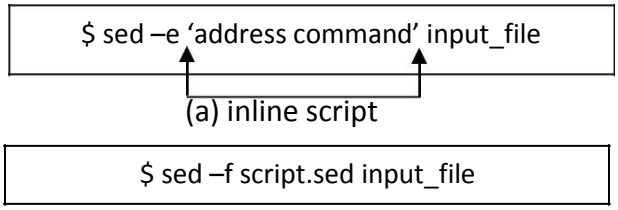
Scripts:

The sed utility is called like any other utility. In addition to input data, sed also requires one or more instructions that provide editing criteria. When there is only one command it may be entered from the keyboard. Most of the time, however, instructions are placed in a file known as sed script (program).

Each instruction in a sed script contains an address and a command.

Script format:

When the script fits in a few lines, its instructions can be included in the command line as shown in figure (a) below. Note that in this case, the script must be enclosed in quotes.



(a) inline script

```
$ sed –f script.sed input_file
```

(b) script file

For longer scripts or for scripts that are going to be executed repeatedly over time, a separate script file is preferred. The file is created with a text editor and saved. In our discussion we suffix the script filename with .sed to indicate that it is a **sed** script. This is not a requirement, but it does make it easier to identify executable scripts. The figure (b) above is an example of executing a sed script.

INSTRUCTION FORMAT:

As previously stated that each instruction consists of an address and a command (figure below :)

The address selects the line to be processed (or not processed) by the command. The exclamation point (!) is an optional address complement. When it is not present, the address must exactly match a line to select the line. When the complement operator is present, any line that does not match the address is selected; lines that match the address are skipped. The command indicates the action that sed is to apply to each input line that matches the address.

Comments:

A comment is a script line that documents or explains one or more instructions in a script. It is provided to assist the reader and is ignored by sed. Comment lines begin with a comment token, which is the pound sign (#). If the comment requires more than one line, each line must start with the comment token.

OPERATION:

Each line in the input file is given a line number by sed. This number can be used to address lines in the text. For each line, sed performs the following operations:

1. Copies an input line to the pattern space. The pattern space is a special buffer capable of holding one or more text lines for processing.

2. Applies all the instructions in the script, one by one, to all pattern space lines that match the specified addresses in the instruction.

3. Copies the contents of the pattern space to the output file unless directed not to by the –n option flag.

When all of the commands have been processed, sed repeats the cycle starting with 1. When you examine this process carefully, you will note that there are two loops in this processing cycle. One loop processes all of the instructions against the current line (operation 2 in the list). The second loop processes all lines.

A second buffer the hold space, is available to temporarily store one or more lines as directed by the sed instructions.

ADDRESSES

The address in an instruction determines which lines in the input file are to be processed by the commands in the instruction. Addresses in sed can be one of four types:
single line, set of lines, range of lines, nested addresses.

Single line Addresses:

A single line address specifies one and only one line in the input file. There are two single-line formats: a line number or a dollar sign ($), which specifies the last line in the input file.

Example:

$4command_1 \rightarrow command_1$ applies only to line 4.

$16command_2 \rightarrow command_2$ applies only to line 16.

$\$command_3 \rightarrow command_3$ applies only to last line.

Set-of-Line Addresses:

A set-of-line address is a regular expression that may match zero or more lines, not necessarily consecutive, in the input file. The regular expression is written between two slashes. Any line in the input file that matches the regular expression is processed by the instruction command.

Two important points need to be noted: First, the regular expression may match several lines that may or may not be consecutive. Second, even if a line matches, the instruction may not affect the line.

Example:

/^A/command$_1$ → Matches all lines that start with "A".

/B$/command$_2$ → Matches all lines that end with "B".

Range Addresses:

An address range defines a set of consecutive lines. Its format is start address, comma with no space, and end address:

start-address,end-address

The start and end address can be a sed line number or a regular expression as in the example below:

line-number,line-number

line-number,/regexp/

/regexp/,line-number

/regexp/,/regexp/

When a line that is in the pattern space matches a start range, it is selected for processing. At this point, sed notes that the instruction is in a range. Each input line is processed by the instruction's command until the stop address matches a line. The line that matches the stop address is also processed by the command, but at that point, the range is no longer active.

If at some future line the start range again matches, the range is again active until a stop address is found. Two important points need to be noted: First, while a range is active, all other instructions are also checked to determine if any of them also match an address.

Second, more than one range may be active at a time. A special case of range address is 1, $, which defines every line from the first line (1) to the last line ($). However, this special case address is not the same as the set-of-lines special case address, which is no address. Given the following two addresses:

(1) Command (2) 1, $command

sed interprets the first as a set of line address and the second as a range address. Some commands, such as insert (i) and append (a), can be used only with a set of line address. These commands accept no address but do not accept 1, $ addresses.

Nested Addresses:

A nested address is an address that is contained within another address. While the outer (first) address range, by definition, must be either a set of lines or an address range, the nested addresses may be either a single line, a set of lines, or another range.

Example-1:

To delete all blank lines between lines 20 and 30

```
20, 30{
        /^$/d
        }
```

The first command specifies the line range; it is the outer command. The second command, which is enclosed in braces, contains the regular expression for a blank line. It contains the nested address.

Example-2:

To delete all lines that contain the word Raven, but only if the line also contains the word Quoth; In this case, the outer address searches for lines containing Raven, while the inner address looks for lines containing Quoth. Here the interesting thing is that the outer address is not a block of lines but a set of lines spread throughout the file.

```
/Raven/{
            /Quoth/d
    }
```

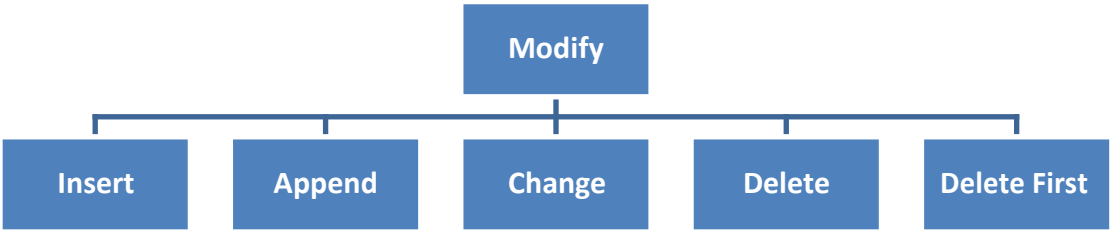
COMMANDS: There are 25 commands that can be used in an instruction. We group them in to nine categories based on how they perform their task. The following figure summarizes the command categories.

LINE NUMBER COMMAND:

The line number command (=) writes the current line number at the beginning of the line when it writes the line to the output without affecting the pattern space. It is similar to the grep –n option. The only difference is that the line number is written on a separate line.

MODIFY COMMAND:

Modify commands are used to insert, append, change, or delete one or more whole lines. The modify commands require that any text associated with them be placed on the next line in the script. Therefore the script must be in a file; it cannot be coded on the shell command line.



In addition, the modify commands operate on the whole line. In other words, they are line replacement commands. This means that we can't use these sed commands to insert text in to the middle of a line. *All modify commands apply to the whole line. You cannot modify just part of a line.*

Insert Command (i):

Insert adds one or more lines directly to the output before the address. This command can only be used with the single line and a set of lines; it cannot be used with a range. If you used the inset command with the all lines address, the lines are inserted before every line in the file. This is an easy way to quickly double space a file.

Append Command (a):

Append is similar to insert command except that it writes the text directly to the output after the specified line. Like insert, append cannot be used with a range address.

Inserted and appended text never appears in sed's pattern space. They are written to the output before the specified line (insert) or after the specified line (append), even if the pattern space is not itself written. Because they are not inserted in to the pattern space, they cannot match a regular expression, nor do they affect sed's internal line counter.

Change Command (c):

Change replaces a matched line with new text. Unlike insert and append, it accepts all four address types.

Delete Pattern Space Command (d):

The delete command comes in two versions. When a lowercase delete command (d) is used, it deletes the entire pattern space. Any script commands following the delete command that also pertain to the deleted text are ignored because the text is no longer in the pattern space.

Delete Only First Line Command (D):

When an uppercase delete command (D) is used, only the first line of the pattern space is deleted. Of course, if the only line in the pattern space, the effect is the same as the lowercase delete.

SUBSTITUTE COMMAND (S):

Pattern substitution is one of the most powerful commands in sed. In general substitute replaces text that is selected by a regular expression with a replacement string. Thus it is similar to the search and replace found in text editors. With it, we can add, delete or change text in one or more lines. The format of substitute command is given below:

Optional

Address	s	/	**Pattern**	/	Replacement String	/	Flag(s)

Search Pattern: The sed search pattern uses only a subset of the regular expression atoms and patterns. The allowable atoms and operators are listed in table below:

Atoms	**Allowed**	**Operators**	**Allowed**
Character	**√**	Sequence	**√**
Dot	**√**	Repetition	* ? \ { . . . \ }
Class	**√**	Alternation	**√**
Anchors	**^ $**	Group	
Back Reference	**√**	Save	**√**

When a text line is selected, its text is matched to the pattern. If matching text is found, it is replaced by the replacement string. The pattern and replacement strings are separated by a triplet of identical delimiters, slashes (/) in the preceding example. Any character can be used as the delimiters, although the slash is the most common.

Replace String:

The replacement text is a string. Only one atom and two metacharacter's can be used in the replacement string. The allowed replacement atom is the back reference. The two metacharacter tokens are the ampersand (&) and the back slash (\). The ampersand is used to place the pattern in the replacement string; the backslash is used to escape an ampersand when it needs to be included in the substitute text (if it's not quoted, it will be replaced by the pattern).

The following example shows how the metacharacter's are used. In the first example, the replacement string becomes ** * * UNIX * * **. In the second example, the replacement string is *now & forever*.

$ **sed 's/UNIX/*** & ***/' file1**

$ **sed '/now/s//now \& forever/' file1**

TRANSFORM COMMAND (Y):

It is sometimes necessary to transform one set of characters to another. For example, IBM mainframe text files are written in a coding system known as Extended Binary Coded Decimal Interchange Code (EBCDIC). In EBCDIC, the binary codes for characters are different from ASCII. To read an EBCDIC file, therefore, all characters must be transformed to their ASCII equivalents as the file is read.

The transform command (y) requires two parallel sets of characters. Each character in the first string represents a value to be changed to its corresponding character in the second string. This concept is presented as shown below:

Address	y	/	Source Characters	/	Replacement Characters	/

As an example to transform lowercase alphabetic characters to their matching uppercase characters, we would make the source set all of the lowercase characters and the replacement set their corresponding uppercase letters. These two sets would transform lowercase alphabetic characters to their uppercase form. Characters that do not match a source character are left unchanged.

INPUT AND OUTPUT COMMANDS:

The sed utility automatically reads text from the input file and writes data to the output file, usually standard output. There are five input/output commands: next (n), append next(N), print (p), print first line(P), and list (l).

Next Command (n):

The next command (n) forces sed to read the next text line from the input file. Before reading the next line, however, it copies the current contents of the pattern space to the output, deletes the current text in the pattern space, and then refills it with the next input line. After reading the input line, it continues processing through the script.

Append Next Command (N):

Whereas the next command clears the pattern space before inputting the next line, append next command (N) does not. Rather, it adds the next input line to the current contents of the pattern space. This is especially useful when we need to apply patterns to two or more lines at the same time.

Print Command (p):

The print command copies the current contents of the pattern space to the standard output file. If there are multiple lines in the pattern space, they are all copied. The contents of the pattern space are not deleted by the print command.

Print First Line Command (P):

Whereas the print command prints the entire contents of the pattern space, the print first line command (P) prints only the first line. That is it prints the contents of the pattern space up to and including a newline character. Anything following the first newline is not printed.

List Command (l):

Depending on the definition of ASCII, there are either 128 (standard ASCII) or 256 (extended ASCII) characters in the character set. Many of these are control characters with no associated graphics. Some, like tab, are control characters that are understood and are actually used for formatting but have no graphic. Others print as spaces because a terminal doesn't support the extended ASCII characters. *The list command (l) converts the unprintable characters to their octal code.*

FILE COMMANDS:

There are two file commands that can be used to read and write files.

Note that there must be exactly one space between the read or write command and the filename. This is one of those sed syntax rules that must be followed exactly.

Read File Command (r):

The read file command reads a file and places its contents in the output before moving to the next command. It is useful when you need to insert one or more common lines after text in a file. The contents of file appear after the current line (pattern space) in the output.

Write File Command:

The write file command (w) writes (actually appends) the contents of the pattern space to a file. It is useful for saving selected data to a file.

Branch Commands:

The branch commands change the regular flow of the commands in the script file. Recall that for every line in the file, sed runs through the script file applying commands that match the current pattern space text. At the end of the script file, the text in the pattern space is copied to the output file, and the next text line is read into the pattern space replacing the old text.

The branch commands allow us to do just that, skip one or more commands in the script file. There are two branch commands: branch (b) and branch on substitution (t).

Branch Label

Each branch command must have a target, which is either a label or the last instruction in the script (a blank label). A label consists of a line that begins with a colon (:) and is followed by up to seven characters that constitute the label name. Example: **:comHere**

Branch Command

The branch command (b) follows the normal instruction format consisting of an address, the command (b) and an attribute (target) that can be used to branch to the end of the script or to a specific location within the script.

The target must be blank or match a script label in the script. If no label is provided, the branch is to the end of the script (after the last line), at which point the current contents of the pattern space are copied to the output file and the script is repeated for the next input line.

Branch on Substitution Command: Rather than branch unconditionally, we may need to branch only if a substitution has been made. In this case we use the branch on substitution or, as it is also known, the test command (t). The format is the same as the branch command.

Hold Space Commands:

The hold buffer is used to save the pattern space. There are five commands that are used to move text back and forth between the pattern space and the hold space: hold and destroy (h), hold and append (H), get and destroy (g), get and append (G) and exchange (x).

Hold and Destroy Command:

The hold and destroy command copies the current contents of the pattern space to the hold space and destroys any text currently in the hold space.

Hold and Append Command:

The hold and append command appends the current contents of the pattern space to the hold space.

Get and Destroy Command:

The get and destroy command copies the text in the hold space to the pattern space and destroy any text currently in the pattern space.

Get and Append Command:

The get and append command appends the current contents of the hold space to the pattern space.

Exchange Command:

The exchange command swaps the text in the pattern and hold space. That is the text in the pattern space is moved to the hold space, and the data that were in the hold space are moved to the pattern space.

Quit: The quit command (q) terminates the sed utility.

awk:

The awk utility which takes its name from the initial of its authors (Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan), is a powerful programming language disguised as a utility. It behavior is to some extent like sed. It reads the input file, line by line, and performs an action on a part of or on the entire line. Unlike sed, however, it does not print the line unless specifically told to print it.

The format of awk is shown below:

awk options script files

Options:

-F: input field separator
-f: script file

EXECUTION:

The awk utility is called like any other utility. In addition to input data, awk also requires one or more instructions that provide editing instructions. When there are only a few instructions, they may be entered at the command line from the keyboard. Most of the time, however, they are placed in a file known as an awk script (program). Each instruction in awk script contains a pattern and an action.

If the script is short and easily fits on one line, it can be coded directly in the command line. When coded on the command line, the script is enclosed in quotes. The format for the command line script is:

$ awk 'pattern[{action}' input -file

For longer scripts or for scripts that are going to be executed repeatedly over time, a separate script file is preferred. To create the script, we use a text editor, such as vi or emacs. Once the script has been created we execute it using the file option (-f), which tells awk that the script is in a file. The following example shows how to execute an awk script:

$ awk –f scriptFile.awk input-file

FIELDS AND RECORDS:

The awk utility vies a file as a collection of fields and records. A field is a unit of data that has informational content. For example, in UNIX list command (ls) output; there are several informational pieces of data, each of which is a field. Among list's output are the permissions, owner, date created, and filename. In awk each field of information is separated from the other fields by one or more whitespace characters or separators defined by the user.

Each line in awk is a record. A record is a collection of fields treated as a unit. In general all of the data in a record should be related. Referring again to the list command output, we see that each record contains data about a file.

When a file is made up of data organized in to records, we call it as a data file, as contrasted with a text file made up of words, lines, and paragraphs. Although awk looks at a

file as a set of records consisting of fields, it can also handle a text file. In this case however, each text line is the record, and the words in the line are fields.

BUFFERS AND VARIABLES:

The awk utility provides two types of buffers: record and field. A buffer is an area of memory that holds data while they are being processed.

Fields Buffers:

There are as many field buffers available as there are fields in the current record of the input file. Each field buffer has a name, which is the dollar sign ($) followed by the field number in the current record. Field numbers begin with one, which gives us $1 (the first field buffer), $2 (the second field buffer) and so on.

Record Buffer:

There is only one record buffer available. Its name is $0. It holds the whole record. In other words, its content is the concatenation of all field buffers with one field separator character between each field.

As long as the contents of any of the fields are not changed, $0 holds exactly the same data as found in the input file. If any fields are changed, however, the contents of the $0, including the field separators, are changed.

Variables:

There are two different types of variables in awk: system variables and user-defined variables.

System Variables:

There are more than twelve system variables used by awk; we discuss some of them here. Their names and function are defined by awk. Four of them are totally controlled by awk. The others have standard defaults that can be changed through a script. The system variables are defined in the table given below:

Variable	Function	Default
FS	Input field separator	Space or tab
RS	Input record separator	Newline
OFS	Output field separator	Space or tab
ORS	Output record separator	Newline
NFa	Number of nonempty fields in current record	
NRa	Number of records read from all files	

Variable	Function	Default
FNR[a]	File number of records read-record number in current file	
FILENAME[a]	Name of the current file	
ARGC	Number of command-line arguments	
ARGV	Command-line argument array	
RLENGTH	Length of string matched by a built-in string function	
RSTART	Start of string matched by a built-in string function	

User Defined Variables:

We can define any number of user defined variables within an awk script. They can be numbers, strings or arrays. Variable names start with a letter and can be followed by any sequence of letters, digits, and underscores. They do not need to be declared; they simply come in to existence the first time they are referenced. All variables are initially created as strings and initialized to a null string (“”).

SCRIPTS:

Data processing programs follows a simple design: preprocessing or initialization, data processing and post processing or end of job. In a similar manner, all awk scripts are divided in to three parts: begin, body, and end (shown in figure below).

BEGIN	{Begin’s Actions}	Preprocessing
	Pattern {Action} Pattern {Action} Pattern {Action}	Body
END	{End’s Actions}	Postprocessing

Initialization Processing (BEGIN): The initialization processing is done only once, before awk starts reading the file. It is identified by the keyword BEGIN, and the instructions are enclosed in a set of branches. The beginning instructions are used to initialize variables, create report headings and perform other processing that must be completed before the file processing starts.

Body Processing: The body is a loop that processes the data in a file. The body starts when awk reads the first record or line from the file. It then processes the data through the body instructions, applying them as appropriate. When end of body instructions is reached, awk repeats the process by reading next record or line & processing it against the body instructions.

End Processing (END): The end processing is executed after all input data have been read. At this time, information accumulated during the processing can be analyzed and printed or other end activities can be conducted.

PATTERNS:

The pattern identifies which records in the file are to receive an action. The awk utility can use several different types of patterns. As it executes a script, it evaluates the patterns against the records found in the file. If the pattern matches the record, the action is taken. If the pattern doesn't match the records, the action is skipped. A statement without a pattern is always true, and the action is always taken. We divide awk patterns in to two categories: simple and range.

Simple Patterns:

A simple pattern matches one record. When a pattern matches a record, the result is true and the action statement is executed. There are four types of simple patterns: BEGIN, END, expression, and nothing (no expression).

BEGIN and END:

BEGIN is true at the beginning of the file before the first record is read. It is used to initialize the script before processing any data; for example it sets the field separators or other system variables. In the below example we set the field separator (FS) and the output field separator (OFS) to tabs.

```
BEGIN
{
FS = "\t" OFS =
"\t"
} # end BEGIN
.
.
.
END
{
printf("Total                Sales:",
totalSales) } # end END
```

END is used at the conclusion of the script. A typical use prints user-defined variables accumulated during the processing.

Expressions:

The awk utility supports four expressions: regular, arithmetic, relational, and logical.

Regular Expressions: The awk regular expressions (regexp) are those defined in egrep. In addition to the expression, awk requires one of two operators: match (~) or not match (!~). When using a regular expression, remember that it must be enclosed in /slashes/.

Example:

```
$0~ /^A.*B$/        # Record must begin with 'A' and end with 'B'
$3 !~ /^ /          # Third field must not start with a space
$4 !~ /bird/        # Fourth field must not contain "bird"
```

Arithmetic Expressions: An arithmetic expression is the result of an arithmetic operation. When the expression is arithmetic, it matches the record when the value is nonzero, either plus or minus; it does not match the record when it is zero (false). The following table list the operators used by awk in arithmetic expressions.

Operator	Example	Explanation
* / % ^	a^2	Variable a is raised to power 2 (a^2)
++	++a, a++	Adds 1 to a.
--	--a, a--	Subtracts 1 from a.
+ -	a + b, a – b	Adds or subtracts two values.
+	+a	Unary plus: value is unchanged
-	-a	Unary minus: value is complemented
=	a = 0	a is assigned the value 0
*=	x *= y	The equivalent of x = x * y
/=	x /= y	The equivalent of x = x / y
%=	x %= y	The equivalent of x = x % y
+=	x += 5	The equivalent of x = x + 5
-=	x -= 5	The equivalent of x = x – 5

Relational Expressions: Relational expressions compare two values and determine if the first is less than, equal to or greater than the second.

Operator	Explanation
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal

Logical Expressions: A logical expression uses logical operator to combine two or more expressions.

Operator	Explanation
!expr	Not expression
$expr_1$ && $expr_2$	Expression 1 and expression 2
$expr_1$ || $expr_2$	Expression 1 or expression 2

Nothing (No Pattern):

When no address pattern is entered, awk applies the action to every line in the input file. This is the easiest way to specify that all lines are to be processed.

Range Patterns:

A range pattern is associated with a range of records or lines. It is made up of two simple patterns separated by a comma as shown here:

start-pattern, end-pattern

The range starts with the record that matches the start pattern and ends with the next record that matches the end pattern. If the start and end patterns are the same, only one record is in the range.

Each simple pattern can be only one expression; the expression cannot be BEGIN or END. If a range pattern matches more than one set of records in the file, then the action is taken for each set. However, the sets cannot overlap. Thus, if the start range occurs twice before the end range, there is only one matching set starting from the first start record through the matching end record. If there is no matching end range, the matching set begins with the matching start record and end with the last record in the file.

ACTIONS:

In programming languages, actions are known as instructions or statements. They are called actions in awk because they act when the pattern is true. Virtually all of the C language capabilities have been incorporated in to awk and behave as they do in C.

In awk an action is one or more statements associated with a pattern. There is a one-to-one relationship between an action and a pattern: One action is associated with only one pattern. The action statements must be enclosed in a set of braces; the braces are required even if there is only one statement. A set of braces containing pattern/action pairs or statements is known as a block. When an action consists of several statements they must be separated by a statement separator.

In awk the statement separators are a semicolon, a newline, or a set of braces (block).

Pattern/Action Syntax:

One Statement Action: pattern {statement}

Multiple Statements Separated by Semicolons: pattern {statement1; statement2; statement3}

Multiple Statements Separated by Newlines:

```
pattern
{
        Statement1
        Statement2
        Statement3
}
```

The awk utility contains a rich set of statements that can solve virtually any programming requirement.

Example to print fields:

$ **awk '{print}' sales.dat**

Output:

```
1    clothing     3141
1    Computers    9161
2    Clothing     3252
```

Example to print selected fields:

$ **awk '{print $1, $2, $3}' sales2.dat | head -2**

Output:

1 clothing 3141
1 computers 9161

UNIT-IV

UNIT-IV

KORN SHELL FEATURES:

The Korn shell, developed by David Korn at the AT&T Labs, is a dual-purpose utility. It can be used interactively as an interpreter that reads, interprets, and executes user commands. It can also be used as a programming language to write shell scripts.

Korn Shell Sessions: When we use the Korn shell interactively, we execute commands at the shell prompt.

Standard Streams: We defined three Standard Streams – standard input (0), standard output (1), and standard error (2) – available in all shells.

Redirection: The standard streams can be redirected from and to files. If we don't use redirection, standard output and standard error both go to the monitor.

Pipes: The pipe operator temporarily saves the output from one command in a buffer that is being used at the same time as the input to the next command.

tee command: The tee command copies standard input to standard output and at the same time copies it to one or more files. If the stream is coming from another command, such as who, it can be piped to the tee command.

Combining Commands: We can combine commands in four ways: sequenced commands, grouped commands, chained commands, and conditional commands.

Command Line Editing: The Korn shell supports command-line editing.

Quotes: There are three quote types that Korn shell supports: backslash, double quotes and single quotes.

Command Substitution: Command Substitution is used to convert a command's output to a string that can be stored in another string or a variable. Although the Korn shell supports two constructs for command substitution ['command' and $ (command)].

Job Control: Job control is used to control how and where a job is executed in the foreground or background.

Aliases:

An alias is a means of creating a customized command by assigning a name or acronym to a command. If the name we use is one of the standard shell commands, such as dir, then the alias replaces the shell command. In the Korn shell, an alias is created by using the **alias command. It's format is: alias name=command-definition**

Where alias is the command keyword, name is the name of the alias name being created, and command-definition is the code for the customized command.

Listing Aliases:

The Korn shell provides a method to list all aliases and to list a specific alias. Both use the alias command. To list all aliases, we use the alias command with no arguments. To list a specific command, we use the alias command with one argument, the name of the alias command.

Removing Aliases:

Aliases are removed by using the **unalias** command. It has one argument, a list of aliases to be removed. When it is used with the all option (-a), it deletes all aliases.

TWO SPECIAL FILES:

There are two special files in UNIX that can be used by any shell.

Trash File (/dev/null):

The trash file is a special file that is used for deleting data. Found under the device (dev) directory, it has a very special characteristic: Its contents are always emptied immediately after receiving data. In other words, no matter how much or how often data are written to it, they are immediately deleted. Physically there is only one trash file in the system: It is owned by the superuser.

Because it is a file, it can be used as both a source and destination. However, when used as a source, the result is always end of the file because it is always empty. While the following two commands are syntactically correct, the first has no effect because the string “Trash me”, when sent to the trash file, is immediately deleted. The second has no effect because the file is always empty, which means that there is nothing to display.

$ **print “Trash me” > /dev/null**

$ **cat /dev/null**

Terminal File (/dev/tty):

Although each terminal in UNIX is a named file, such as /dev/tty13 and /dev/tty31, there is only one logical file, /dev/tty. This file is found under the device directory; it represents the terminal of each user. This means that someone using terminal /dev/tty13 can refer to the terminal using either the full terminal name (/dev/tty13) or the generic system name (/dev/tty).

VARIABLES:

The Korn shell allows you to store the values in variables. A shell variable is a location in memory where values can be stored. In the Korn shell, all data are stored as strings. There are two broad classifications of variables: user-defined and predefined.

User-Defined Variables:

As implied by their name, user defined variables are created by the user. Although the user may choose any name, it should not be the same as one of the predefined variables. Each variable must have a name. The name of the variable must start with an alphabetic or underscore (_) character. It then can be followed by zero or more alphanumeric or underscore characters.

Predefined Variables:

Predefined variables are either shell variables or environmental variables. The shell variables are used to configure the shell. The environmental variables are used to configure the environment.

Storing Values in Variables:

There are several ways that we can store a value in a variable, but the easiest method is to use the assignment operator, =. The variable is coded first, on the left, followed by the assignment operator and then the value to be stored. There can be no spaces before and after the assignment operator; the variable, the operator, and the value must be coded in sequence immediately next to each other as ***varA=7*** Here varA is the variable that receives the data, and 7 is the value being stored in it.

Accessing Value of a Variable: To access the value of variable, the name of the variable must be preceded by a dollar sign as shown below:

$ **count=7**
$ **print *$count* is the number after 6 and before 8**
Result:
7 is the number after 6 and before 8

Null Variables:

If we access a variable that is not set (no value is stored in it), we receive what is called a null value (nothing). We can also explicitly store a null value in a variable by either assigning it a null string (“”) or by assigning it nothing.

Unsetting a Variable:

We can clear a variable by assigning a null value to it. Although this method works, it is better to use the **unset** command.

$ **x=1**
$ **print "(x contains:" $x")"**
(x contains: 1)

$ **unset x**
$ **print "(x contains:" $x")"**
(x contains:)

Storing Filenames:

We can also store a filename in a variable. We can even use the wildcards. However, we should be aware of how wildcards are handled by the shell. The shell stores the file name including the wildcard in the variable without expanding it. When the value is used, the expansion takes place.

Example:

$ **ls**
File1 File2 File3.bak
$ **filename="File*"**
$ **print "Filename contains: $filename"** # show contents
Filename contains: File*
$ **print $filename**
File1 File2 File3.bak

$ **filename="File?"**
$ **print $filename**
File1 File2

Storing File Contents:

We can also store the contents of a file in a variable for processing, such as parsing words. Two steps are required to store the file:

1. Create a copy of the file on standard output using the **cat** utility.

2. Using command substitution, convert the standard output contents to a string.

The string can now be stored in a variable. The entire process is done in one command line.

Example:

$ **cat storeAsVar.txt**
This is a file
used to show
the result of storing a file in a variable

$ **x=$(cat storeAsVar.txt)**
$ **print $x**
This is a file used to show the result of storing a file in a variable

Storing Commands in a Variable:

We can also store a command in a variable. For example, the list command can be stored in a variable. We can then use the variable at the command prompt to execute its contents. Storing commands in a variable works only with simple commands. If the command is complex (for example, piping the results of the list command to **more**) a command variable will not work.

Read-Only Variables:

Most programming languages provide a way for a programmer to define a named constant. 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 command format is:

readonly variable-list

Example:

$ **cHello=Hello**
$ **cBye="Good Bye"**
$ **readonly cHello cBye**
$ **cHello=Howdy**
cHello: is read only
$ **cBye=TaTa**
cBye: is read only
$ **print cHello " . . . " $cBye**
Hello . . . Good Bye

INPUT AND OUTPUT:

INPUT:

Reading data from a terminal or a file is done using the **read** command. The **read** command reads a line and stores the words in variables. It must be terminated by a return, and the input line must immediately follow the command. The **read** command format is shown below:

read options variable$_1$ . . . variable$_n$

Options:

-r: ignore newline -u:
stream descriptor

Read Word by Word:

When the read command is executed, the shell reads a line from the standard input and stores it in variables word by word. Words are characters separated by spaces or tabs. The first word is stored in the first variable; the second is stored in the second variable, and so forth. Another way of saying this is that the read command parses the input string (line) into words.

If there are more words than there are variables, all the extra words are placed in the last variable. If there are fewer words then there are variables; the unmatched variables are set to a null value. Any value in them before the read is lost.

Reading Line by Line:

The design for handling extra words provides an easy technique for storing a whole line in one variable. We simply use the read command, giving it only one variable. When executed, the whole line is in the variable.

Reading from a File:

The Korn shell allows scripts to read from a user file. This is done with the stream descriptor option (-u). A stream descriptor is a numeric designator for a file. We have seen that the standard streams are numbered 0, 1, and 2 for standard input, standard output and standard error respectively.

OUTPUT:

The output statement in the Korn shell is the print command. Although the Korn shell also supports the echo command (inherited from the Bourne shell), we use print because it is

faster and there is the possibility that echo may become depreciated in a future version of Korn shell. The format of the print command is shown below:

print options argument$_1$ . . . argument$_n$

Options:

-n : no new line

ENVIRONMENT VARIABLES:

The environmental variables control the user environment. The following table lists the environmental variables. In Korn shell, environmental variables are in uppercase.

VARIABLE	EXPLANATION
CDPATH	Contains the search path for cd command when the directory argument is a relative path name.
COLUMNS	Defines the width, in characters, of your terminal. The default is 80.
EDITOR	Pathname of the command-line editor.
ENV	Pathname of the environment file.
HISTFILE	Pathname for the history file.
HISTSIZE	Maximum number of saved commands in the history file.
HOME	Pathname for the home directory.
LINES	Defines the height, in lines, of your terminal display. The default is 24.
LOGNAME	Contains the user's login name from the /etc/passwd file
MAIL	Absolute pathname for the user's mailbox.
MAILCHECK	Interval between tests for new mail. The default is 600 seconds.
OLDPWD	Absolute pathname of the working directory before the last cd command.
PATH	Searches path for commands.
PS1	Primary prompt, such as $ and %.

STARTUP SCRIPTS:

Each shell uses one or more scripts to initialize the environment when a session is started. The Korn shell uses three startup files. They are 1) System profile file 2) Personal profile file and 3) Environment file.

SYSTEM PROFILE FILE:

The system-level profile file is a one which is stored in the /etc directory. Maintained by the system administrator, it contains general commands and variable settings that are applied to every user of the system at login time. The system profile file is generally quite large and contains many advanced commands.

The system profile is a read-only file; its permissions are set so that only system administrator can change it.

PERSONAL PROFILE FILE:

The personal profile, *~/.profile*, contains commands that are used to customize the startup shell. It is an optional file that is run immediately after the system profile file. Although it is a user file, it is often created by the system administrator to customize a new user's shell. If you make changes to it, we highly recommend that you make a backup copy first so that it may be restored easily if necessary.

ENVIRONMENT FILE:

The Korn shell allows users to create a command file containing commands that they want to be executed to personalize their environment. It is most useful when the Korn shell is started as a child of a non-Korn login shell. Because we can use any name for it, the absolute pathname of the environment file must be stored in the ENV variable. The shell then locates it by looking at the ENV variable.

COMMAND HISTORY:

The Korn shell provides an extensive command history capability consisting of a combination of commands, environmental variables and files. A major feature of the design is the ability to recall a command and reexecute it without typing it again.

HISTORY FILE:

Every command that we type is kept in a history file stored in our home directory. By default, the filename is ~/ .sh_history. It can be renamed, provided that we store its pathname in the HISTFILE environmental variable.

The size of the file (i.e. the number of commands that it can store) is 128 unless changed. The HISTSIZE variable can be used to change it when we need to make it larger or smaller.

HISTORY COMMAND:

The formal command for listing, editing, and executing commands from the history file is the fc command. However, the Korn shell contains a preset alias, history, that is easier to use and more flexible. Executed without any options, the history command lists the last 16 commands.

REDO COMMAND (r):

Any command in the history file can be reexecuted using the redo command (r).

SUBSTITUTION IN REDO COMMAND:

When we redo a command, we can change part of the command.

COMMAND EXECUTION PROCESS

To understand the behavior of the shell, it helps to understand how Korn executes a command. Command execution is carried out in six sequential steps:

EXECUTION STEPS:

The six execution steps are recursive. This means that when the shell performs the third step, command substitution, the six steps are followed for the command inside the dollar parentheses.

Command Parsing: The shell first parses the command into words. In this step, it uses whitespace as delimiters between the words. It also replaces sequences of two or more spaces or tabs with a single space.

Variable Evaluation: After completely parsing the command, the shell looks for variable names (unquoted words beginning with a dollar sign). When a variable name is found, its value replaces the variable name.

Command Substitution: The shell then looks for a command substitution. If found, the command is executed and its output string replaces the command, the dollar sign, and the parenthesis.

Redirection: At this point, the shell checks the command for redirected files. Each redirected file is verified by opening it.

Wildcard Expansion: When filenames contain wildcards, the shell expands and replaces them with their matching filenames. This step creates a file list.

Path Determination: In this last step, the shell uses the PATH variable to locate the directory containing the command code. The command is now ready for execution.

KORN SHELL PROGRAMMING:

Basic Script Concepts:

A shell script is a text file that contains executable commands. 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.

Script Components:

Every script has three parts: the interpreter designator line, comments and shell commands.

Interpreter Designator Line: One of the UNIX shells runs the script, reading it and calling the command specified by each line in turn. The first line of the script is the designator line; it tells UNIX the path to the appropriate shell interpreter. The designator line begins with a pound sign and a bang (**#!**). If the designator line is omitted, UNIX will use the interpreter for the current shell, which may not be correct.

Comments: Comments are documentation we add in a script to help us understand it. The interpreter doesn't use them at all; it simply skips over them.

Comments are identified with the pound sign token (#). The Korn shell supports only line comments. This means that we can only comment one line at a time, c comment cannot extend beyond the end of the line.

Commands: The most important part of a script is its commands. We can use any of the commands available in UNIX. However, they will not be executed until we execute the script; they are not executed immediately as they are when we use them interactively. When the script is executed, each command is executed in order from the first to the last.

Command Separators → Shell use two tokens to separate commands; semicolons and newlines.

Blank Lines → Command separators can be repeated. When the script detects multiple separators, it considers them just one. This means that we can insert multiple blank lines in a script to make it more readable.

Combined Commands We can combine commands in a script just as we did in the interactive sessions. This means that we can chain commands using pipes, group commands or conditional commands.

MAKING SCRIPTS EXECUTABLE:

We can make a script executable only by the user (ourselves), our group, or everybody. Because we have to test a new script, we always give ourselves execute permission. Whether or not we want others to execute it depends on many factors.

EXECUTING THE SCRIPT:

After the script has been made executable, it is a command and can be executed just like any other command. There are two methods of executing it; as an independent command or as an argument to a subshell command.

Independent Command:

We do not need to be in the Korn shell to execute a Korn shell script as long as the interpreter designator line is included as the first line of the script. When it is, UNIX uses the appropriate interpreter as called out by the designator line.

To execute the script as an independent command, we simply use its name as in the following example:

$ **script_name**

Child Shell Execution:

To ensure that the script is properly executes, we can create a child shell and execute it in the new shell. This is done by specifying the shell before the script name as in the following example:

$ **ksh script_name**

EXPRESSIONS:

Expressions are a sequence of operators and operands that reduces to a single value. The operators can be either mathematical operator, such as add and subtract, that compute a value; relational operators, such as greater than and less than, that determine a relationship between two values and return true or false; file test operators that report status of a file; or logical operators that combine logical values and return true or false. We use mathematical expressions to compute a value and other expressions to make decisions.

Mathematical expressions

Mathematical expressions in the Korn shell use integer operands and mathematical operators to compute a value.

Mathematical operators

Mathematical operators are used to compute a numeric value. The Korn shell supports the standards add, subtract, multiply and divide operators plus a special operator for modulus.

let command:

The Korn shell uses either the expr command or the let command to evaluate expressions and store the result in another variable. The expr command is inherited from the Bourne shell; the let command is new.

Example:

$ **let y=x+16**

In this example, note that we don't use a dollar sign with the variables. The **let** command doesn't need the dollar sign; its syntax expects variables or constants.

The Korn shell has an alternate operator, a set of double parentheses, that may be used instead of let command.

Example:

$ **((y = x + 16))**

Relational expressions:

It compares two values and returns a logical value such as true or false. The logical value depends on the values being compared and the operator being used.

Relational operators:

The relational operators are listed in table given below:

Numeric Interpretation	Meaning	String Interpretation
>	Greater than	
>=	Greater than or equal	
<	Less than	
<=	Less than or equal	
==	Equal	=

Numeric Interpretation	Meaning	String Interpretation
!=	Not equal	!=
	String length not zero	-n
	String length zero	-z

The sting equal and not equal logical operators support patterns for the second (right) operand. The patterns supported are listed in the table below:

Pattern	Interpretation
String	Must exactly match the first operand.
?	Matches zero or one single character.
[. . .]	Matches one single character in the set.
*	Repeats pattern zero or more times.
?(pat1|pat2|. . .)	Matches zero or one of any of the patterns.

Relational Test Command:

In the Korn shell we can use wither the test command inherited from the Bourne shell or one of the two test operators, ((. . .)) or [[. . .]].

Which operator is used depends on the data. Integer data require the double parenthesis as shown in the example: i.e. **((x < y))**

For string expressions, the Korn shell requires the double bracket operator. Although the integer operator parentheses do not require the variable dollar sign, the double brackets operator does. The next example demonstrates this format:

[[$x != $y]]

File Expressions:

File expressions use file operators and test command to check the status of a file. A file's status includes characteristics such as open, readable, writable, or executable.

File Operators:

There are several operators that can be used in a file test command to check a files status. They are particularly useful in shell scripts when we need to know the type or status of file. The following table lists the file operators and what file attributes they test.

Operator	Explanation
-r file	True if file exists and is readable
-l file	True if file exists and is a symbolic link.

Operator	Explanation
-w file	True if file exists and is writable.
-x file	True if file exists and is executable.
-f file	True if file exists and is a regular file.
-d file	True if file exists and is a directory.
-s file	True if file exists and has a size greater than zero.
file1 –nt file2	True if file1 is newer than file2.
file1 –ot file2	True if file1 is older than file2.

Test File Command:

Although we could use the test command inherited from the Bourne shell, in the Korn shell we recommend the Korn shell double bracket operator to test the status of the file.

Logical Expressions:

Logical expressions evaluate to either true or false. They use a set of three logical operators: not (!), and (&&), or (||).

DECISION MAKING & REPETITION:

DECISION MAKING:

The Korn shell has two different statements that allow us to select between two or more alternatives. The first, the if-then-else statement, examines the data and chooses between two alternatives. For this reason this is sometimes referred to as a two-way selection. The second, the case statement, selects one of several paths by matching patterns to different strings.

if-then-else

Every language has some variation of the if-then-else statement. The only difference between them is what keywords are required by the syntax. For example, the C language does not use then. In fact, it is an error to use it. In all languages however, something is tested. Most typically data values are tested.

In the Korn shell, the exit value from a command is used as the test. The shell evaluates the exit status from the command following fi. When the exit status is 0, the then set of commands is executed. When the exit status is 1, the else set of commands is executed.

Case syntax: The case statement contains the string that is evaluated. It ends with an end case token, which is esac (case spelled backward). Between the start and end case statements is the **pattern list.**

For every pattern that needs to be tested, a separate patter is defined in the pattern list. The pattern ends with a closing parenthesis. Associated with each pattern is one or more commands. The commands follow the normal rules for commands with the addition that the last command must end in two semicolons. The last action in the pattern list is usually the wildcard asterisk, making it the default if none of the other cases match.

REPETITION:

The real power of computers is their ability to repeat an operation or a series of operations many times. This repetition, known as looping, is one of the basic programming concepts.

A loop is an action or a series of actions repeated under the control of loop criteria written by the programmer. Each loop tests the criteria. If the criteria tests valid, the loop continues; if it tests invalid, the loop terminates.

Command-Controlled and List-Controlled Loops:

Loops in Korn shell can be grouped into two general categories: command-controlled loops and list-controlled loops.

Command-Controlled loops:

In a command controlled loop, the execution of a command determines whether the loop body executes or not. There are two command-controlled loops in the Korn shell : the *while* loop and the *until* loop.

Syntax for while:

```
while
command do
        action
done
```

The until loop works just like while loop, except that it loops as long as the exit status of the command is false. In this sense, it is the complement of the while loop. The syntax of until loop is:

```
until command
do
        action
done
```

List-Controlled loops:

In a list-controlled loop, there is a control list. The number of elements in the list controls the number of iterations. If the control list contains five elements, the body of the loop is executed five times; if the control list contains ten elements, the body of the loop is executed ten times.

The **for-in** loop is the first list-controlled loop in the Korn shell. The list can be any type of string; for example it can be words, lines, sentences, or a file.

for-in loop Syntax:

for variable in list
do
    action
done

Example:

```
for I in 1 2 3 4 5
do
        print $I hello
done
```

The **select** loop is the second Korn shell list-controlled loop. The select loop is a special loop designed to create menus. A menu is a list of options displayed on the monitor. The user selects one of the menu options, which is then processed by the script. The format of the select loop is similar to for-in loop. It begins with the keyword select followed by a variable and a list of strings:

$ select variable in list

Example:

```
select choice in month year quit
do
        case $choice in
                month) cal; ;
                year)   yr-$(date "+%Y")
                        cal $yr; ;
                quit)   print "Hope you found your date"
                        exit; ;
                *)      print "sorry, I don't understand your answer"
        esac
done
```

SPECIAL PARAMETERS AND VARIABLES

The Korn shell provides both special parameters and special variables for our use.

SPECIAL PARAMETERS:

Besides having positional parameters numbered 1 to 9, the Korn shell script can have four other special parameters: one that contains the script filename, one that contains the number of arguments entered by the user, and two that combine all other parameters.

Script Name ($0):

The script name parameter ($0) holds the name of the script. This is often useful when a script calls other script. The script name parameter can be passed to the called script so that it knows who called it. As another use, when a script needs to issue an error message, it can include its name as part of the message. Having the script name in the message clearly identifies which script had a problem.

Number of Arguments ($#):

A second special parameter holds the number of arguments passed to the script. Scripts can use this parameter, referred to as $#, programmatically in several ways.

All Parameters ($* and $@):

Two special parameters combine the nine positional parameters in to one string. They can be used with or without quotes.

SPECIAL VARIABLES:

Internal Field Separator (IFS):

The IFS variable holds the tokens used by the shell commands to parse the string into substrings such as words. The default tokens are the three white space tokens: the space, tab and newline.

One common use of the internal field separators parses a read string into separate words. It receives a login id as an argument and then searches the password file (/etc/passwd) for the matching id. When it finds, it prints the login id and the user name.

Special Parameter and Variable Summary:

Parameter or Variable	Description
$#	Number of arguments to a script

Parameter or Variable	Description
$0	Script Name
$*	All parameters
$@	All parameters
$?	Exit status variable
IFS	Internal field separator

CHANGING POSITIONAL PARAMETERS:

The positional parameters can be changed within the script only by using the set command; values cannot be directly assigned to them. This means that to assign values to the script positional parameters, we must use set. The set command parses an input string and places each separate part of the string into a different positional parameter (up to nine).

If the IFS is set to the default, set parses to words. We can set the IFS to any desired token and use set to parse the data accordingly.

Shift Command:

One very useful command in shell scripting is the shift command. The shift command moves the values in the parameters toward the beginning of the parameter list. To understand the shift command, think of the parameters as a list rather than as individual variables. When we shift, we move each parameter to the left in the list. If we shift three positions, therefore, the fourth parameter will be in the first position, the fifth will be in the second position, and so forth until all parameters have been moved three positions to the left. The parameters at the end of the set become null.

ARGUMENT VALIDATION:

Good programs are designed to be fail-safe. This means that anything that can be validated should be confirmed before it is used. We discuss various techniques used to validate user-supplied arguments:

Number of Arguments Validation:

The first code in a script that contains parameters should validate the number of arguments. Some scripts use a fixed number of arguments; other scripts use a variable number of arguments.

Even when the number of arguments is variable, there is usually a minimum number that is required. Both fixed-and variable- numbered arguments are validated by using the number of arguments parameter ($#).

Minimum Number of Arguments:

When a script expects a variable number of arguments and there is a minimum number required, we should verify that the minimum number has been entered.

Type of Argument Validation:

After the exact or minimum number of arguments is validated, the script should verify that each arguments type is correct. While all arguments are passed as strings, the string contents can be a number, a filename, or any other verifiable type.

Numeric Validation:

The value of numeric parameters is virtually unlimited; some scripts simply need number. Scripts that extract a range of lines from a file are of this nature. Other scripts may require that the number be in a range.

File Type Validation:

If an argument is an input file, we can verify that the file exists and that it has read permission. If the file is an output file, there is no need to verify it because UNIX will create it if it doesn't exist.

DEBUGGING SCRIPTS:

Whenever we write a script we test it. Often multiple tests are necessary. Sometimes the tests don't deliver the expected results. In these cases, we need to debug the script. There are two Korn shell options that we can use to help debug script: the verbosity (verbose) option and the execute trace (xtrace) option.

The *verbose* option prints each statement that is syntactically correct and displays an error message if it is wrong. Script output, if any is generated.

The xtrace option prints each command, preceded by a plus (+) sign, before it is executed. It also replaces the value of each variable accessed in the statement. For example, in the statement y=$x, the $x is replaced with actual variable value at the time the statement is executed.