6

Basic File Attributes

In the previous two chapters, you created files and directories, navigated the file system, and copied, moved and removed files without any problem. In real life, however, matters may not be so rosy. You may have problems when handling a file or directory. Your file may be modified or even deleted by others. A restoration from a backup may be unable to write to your directory. You must know why these problems happen and how to rectify and prevent them.

The UNIX file system lets users access other files not belonging to them and without infringing on security. A file also has a number of *attributes* (properties) that are stored in the inode. In this chapter, we'll use the `ls -l` command with additional options to display these attributes. We'll mainly consider the two basic attributes—permissions and ownership—both of which are changeable by well-defined rules. The remaining attributes are taken up in Chapter 11.

What You Will Learn

- Interpret the significance of the seven fields of the `ls -l` output (*listing*).
- How to obtain the listing of a specific directory.
- The importance of *ownership* and *group ownership* of a file and how they affect security.
- The significance of the nine permissions of a file as applied to different *categories* of users.
- Use **chmod** to change all file permissions in a relative and absolute manner.
- Use **chown** and **chgrp** to change the owner and group owner of files on BSD and AT&T systems.

Topics of Special Interest

- The importance of directory permissions (discussion on individual permissions deferred to Chapter 11).
- An introductory treatment of the relationship that exists between file ownership, file permissions and directory permissions.

6.1 ls -l: Listing File Attributes

We have already used the **ls** command with a number of options. It's the `-l` (long) option that reveals most. This option displays most attributes of a file—like its permissions, size and ownership

details. The output in UNIX lingo is often referred to as the **listing**. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence.

ls looks up the file's inode to fetch its attributes. Let's use `ls -l` to list seven attributes of all files in the current directory:

```
$ ls -l
total 72
-rw-r--r--   1 kumar    metal     19514 May 10 13:45 chap01
-rw-r--r--   1 kumar    metal      4174 May 10 15:01 chap02
-rw-rw-rw-   1 kumar    metal        84 Feb 12 12:30 dept.lst
-rw-r--r--   1 kumar    metal      9156 Mar 12  1999 genie.sh
drwxr-xr-x   2 kumar    metal       512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal       512 May  9 09:57 progs
```

The list is preceded by the words `total 72`, which indicates that a total of 72 blocks are occupied by these files on disk, each block consisting of 512 bytes (1024 in Linux). We'll now briefly describe the significance of each field in the output.

File Type and Permissions The first column shows the type and *permissions* associated with each file. The first character in this column is mostly a `-`, which indicates that the file is an ordinary one. This is, however, not so for the directories `helpdir` and `progs`, where you see a `d` at the same position.

You then see a series of characters that can take the values `r`, `w`, `x` and `-`. In the UNIX system, a file can have three types of permissions—read, write and execute. You'll see how to interpret these permissions and also how to change them in Sections 6.4 and 6.5.

Links The second column indicates the number of *links* associated with the file. This is actually the number of filenames maintained by the system of that file. UNIX lets a file have as many names as you want it to have, even though there is a single file on disk. This attribute will be discussed in Chapter 11.

Note: A link count greater than one indicates that the file has more than one name. That doesn't mean that there are two copies of the file.

Ownership When you create a file, you automatically become its *owner*. The third column shows kumar as the owner of all of these files. The owner has full authority to tamper with a file's contents and permissions—a privilege not available with others except the root user. Similarly, you can create, modify or remove files in a directory if you are the owner of the directory.

Group Ownership When opening a user account, the system administrator also assigns the user to some group. The fourth column represents the *group owner* of the file. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file. It's generally desirable that the group have a set of privileges distinct from others as well as the owner. Ownership and group ownership are elaborated in Section 6.7.

Scanned by CamScanner

File Size The fifth column shows the size of the file in bytes, i.e., the amount of data it conta
The important thing to remember is that it is only a character count of the file and not a me
of disk space that it occupies. The space occupied by a file on disk is usually larger than this fig
since files are written to disk in blocks of 1024 bytes or more. In other words, even though the
dept.lst contains 84 bytes, it would occupy 1024 bytes on disk on systems that use a block siz
1024 bytes. We'll discuss the significance of block size much later in the text.

The two directories show smaller file sizes (512 bytes each). This is to be expected as a direc
maintains a list of filenames along with an identification number (the inode number) for each f
The size of the directory file depends on the size of this list—whatever be the size of the f
themselves.

Last Modification Time The sixth, seventh and eighth columns indicate the last modifica
time of the file, which is stored to the nearest second. A file is said to be modified only i
contents have changed in any way. If you change only the permissions or ownership of
file, the modification time remains unchanged. If the file is less than a year old since its l
modification date, the year won't be displayed. Note that the file genie.sh has been modified m
than a year ago.

You'll often need to run automated tools that make decisions based on a file's modification ti
This column shows two other time stamps when ls is used with other options. The time sta
are discussed in Chapter 11.

Filename The last column displays the filenames arranged in ASCII collating sequen
You already know *(4.2)* that UNIX filenames can be very long (up to 255 characters). If y
would like to see an important file at the top of the listing, then choose its name in uppercase
at least, its first letter.

The order of the list can be changed by combining the -l option with other options. In the rest
the chapter, we'll discuss permissions and ownership, and also learn how to change them.

6.2 THE -d OPTION: LISTING DIRECTORY ATTRIBUTES

You'll recall *(4.11.1)* that ls, when used with directory names, lists files in the directory rather tha
the directory itself. To force ls to list the attributes of a directory, rather than its contents, you nee
to use the -d (directory) option:

```
$ ls -ld helpdir progs
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

Directories are easily identified in the listing by the first character of the first column, which be
shows a d. For ordinary files, this slot always shows a - (hyphen), and for device files, either a b
c. The significance of the attributes of a directory differ a good deal from an ordinary file. Directorie
will be considered in some detail in Chapter 11.

Tip: To see the attributes of a directory rather than the files contained in it, use ls -ld with the directory name. Note that simply using ls -d will *not* list all subdirectories in the current directory. Strange though it may seem, ls has no option to list only directories!

6.3 FILE OWNERSHIP

Before we take up file permissions, you need to understand the significance of file ownership. When you create a file, your username shows up in the third column of the file's listing; you are the **owner** of the file. Your group name is seen in the fourth column; your group is the **group owner** of the file. If you copy someone else's file, you are the owner of the copy. If you can't create files in other users' home directories, it's because those directories are not owned by you (and the owner has not allowed you write access).

Several users may belong to a single group. People working on a project are generally assigned a common group, and all files created by group members (who have separate user-ids) will have the same group owner. However, make no mistake: The privileges of the group are set by the owner of the file and not by the group members.

When the system administrator creates a user account, he has to assign these parameters to the user:

- The user-id (UID)—both its name and numeric representation.
- The group-id (GID)—both its name and numeric representation.

The file /etc/passwd maintains the UID (both the number and name) and GID (but only the number). /etc/group contains the GID (both number and name). We'll discuss these two files when learning to add a user account in Chapter 15.

Tip: To know your own UID and GID without viewing /etc/passwd and /etc/group, use the id command:

```
$ id
uid=655537(kumar) gid=655535(metal)
```

The uid is kumar, and if this name matches the one shown in the ownership field of the listing (ls -l output), then you are the owner of those files. It also means that you have the authority to change the file's attributes.

6.4 FILE PERMISSIONS

UNIX has a simple and well-defined system of assigning permissions to files. Let's issue the ls -l command once again to view the permissions of a few files:

```
$ ls -l chap02 dept.lst dateval.sh
-rwxr-xr--   1 kumar    metal       20500 May 10 19:21 chap02
-rwxr-xr-x   1 kumar    metal         890 Jan 29 23:17 dateval.sh
-rw-rw-rw-   1 kumar    metal          84 Feb 12 12:30 dept.lst
```

Scanned by CamScanner

Observe the first column that represents file permissions. These permissions are also different for the three files. UNIX follows a three-tiered file protection system that determines a file's access rights. To understand how this system works, let's break up the permissions string of the file chap02 into three groups. The initial - (in the first column) represents an ordinary file and is left out of the permissions string.

```
rwx   r-x   r--
```

Each group here represents a **category** and contains three slots, representing the read, write and execute permissions of the file—in that order. r indicates read permission, which means **cat** can display the file. w indicates write permission; you can edit such a file with an editor. x indicates execute permission; the file can be executed as a program. The - shows the absence of the corresponding permission.

The first group (rwx) has all three permissions. The file is readable, writable and executable by the *owner* of the file, kumar. But do we know who the owner is? Yes we do. The third column shows kumar as the owner and the first permissions group applies to kumar. You have to log in with the username kumar for these privileges to apply to you.

The second group (r-x) has a hyphen in the middle slot, which indicates the absence of write permission by the *group owner* of the file. This group owner is metal, and all users belonging to the metal group have read and execute permissions only.

The third group (r--) has the write and execute bits absent. This set of permissions is applicable to *others*, i.e., those who are neither the owner kumar nor belong to the metal group. This category (others) is often referred to as the *world*. This file is not world-writable.

You can set different permissions for the three categories of users—owner, group and others. It's important that you understand them because a little learning here can be a dangerous thing. A faulty file permission is a sure recipe for disaster.

Note: The group permissions here don't apply to kumar (the owner) even if kumar belongs to the metal group. The owner has its own set of permissions that override the group owner's permissions. However, when kumar renounces the ownership of the file, the group permissions then apply to him.

6.5 chmod: CHANGING FILE PERMISSIONS

Before we take up **chmod**, let's decide to change a habit. Henceforth, we'll refer to the owner as *user* because that's how the **chmod** command (which changes file permissions) refers to the owner. In this section, whenever we use the term *user*, we'll actually mean *owner*.

A file or directory is created with a default set of permissions, and this default is determined by a simple setting (called *umask*), which we'll discuss later. Generally, the default setting write-protects a file from all except the user (new name for owner), though all users *may* have read access. However, this may not be so on your system. To know your system's default, create a file **xstart**:


```
$ cat /usr/bin/startx > xstart                    Actually copies the file startx
$ ls -l xstart
-rw-r--r--   1 kumar    metal        1906 Sep  5 23:38 xstart
```

It seems that, by default, a file doesn't also have execute permission. So how does one execute such a file? To do that, the permissions of the file need to be changed. This is done with **chmod**.

The **chmod** (change mode) command is used to set the permissions of one or more files for all three categories of users (user, group and others). It can be run only by the user (the owner) and the superuser. The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions.
- In an absolute manner by specifying the final permissions.

We'll consider both ways of using **chmod** in the following sections.

6.5.1 Relative Permissions

When changing permissions in a relative manner, **chmod** only changes the permissions specified in the command line *and leaves the other permissions unchanged*. In this mode it uses the following syntax:

```
chmod category operation permission filename(s)
```

chmod takes as its argument an expression comprising some letters and symbols that completely describe the user category and the type of permission being assigned or removed. The expression contains three components:

- User *category* (user, group, others)
- The *operation* to be performed (assign or remove a permission)
- The type of *permission* (read, write, execute)

By using suitable abbreviations for each of these components, you can frame a compact expression and then use it as an argument to **chmod**. The abbreviations used for these three components are shown in Table 6.1.

Now let's consider an example. To assign execute permission to the user (We won't remind again that user here is the owner.) of the file **xstart**, we need to frame a suitable expression by using appropriate characters from each of the three columns of Table 6.1. Since the file needs to be executable only by the user, the expression required is u+x:

```
$ chmod u+x xstart
$ ls -l xstart
-rwxr--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged. You can now execute the file if you are the owner of the file but the other categories (i.e., group and others) still can't. To enable all of them to execute this file, you have to use multiple characters to represent the user category (ugo):

Scanned by CamScanner

```
$ chmod ugo+x xstart ; ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
```

The string ugo combines all the three categories—user, group and others. UNIX also offers a shorthand symbol a (all) to act as a synonym for the string. And, as if that wasn't enough, there's an even shorter form that combines these three categories. When it is not specified, the permission applies to all categories. So the previous sequence can be replaced by either of the following:

```
chmod a+x xstart
chmod +x xstart
```

a implies ugo
By default, a is implied

chmod accepts multiple filenames in the command line. When you need to assign the same set of permissions to a group of files, this is what you should do:

```
chmod u+x note note1 note3
```

Permissions are removed with the - operator. To remove the read permission from both group and others, use the expression go-r:

```
$ ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
$ chmod go-r xstart ; ls -l xstart
-rwx--x--x   1 kumar    metal        1906 May 10 20:30 xstart
```

chmod also accepts multiple expressions delimited by commas. For instance, to restore the original permissions to the file xstart, you have to remove the execute permission from all (a-x) and assign read permission to group and others (go+r):

```
$ chmod a-x,go+r xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

More than one permission can also be set; **u+rwx** is a valid **chmod** expression. So setting write and execute permissions for others is no problem:

```
$ chmod o+wx xstart ; ls -l xstart
-rw-r--rwx   1 kumar    metal        1906 May 10 20:30 xstart
```

We described relative permissions here, but **chmod** also uses an absolute assignment system, which is taken up in the next topic.

Table 6.1 Abbreviations Used by **chmod**

Category	*Operation*	*Permission*
u—User	+—Assigns permission	r—Read permission
g—Group	-—Removes permission	w—Write permission
o—Others	=—Assigns absolute permission	x—Execute permission
a—All (ugo)		

6.5.2 Absolute Permissions

Sometimes you don't need to know what a file's current permissions are, but want to set all nine permission bits explicitly. The expression used by **chmod** here is a string of three octal numbers (base 8). You may or may not be familiar with this numbering system, so a discussion would be in order.

Octal numbers use the base 8, and octal digits have the values 0 to 7. This means that a set of three bits can represent one octal digit. If we represent the permissions of each category by one octal digit, this is how the permissions can be represented:

- Read permission—4 (Octal 100)
- Write permission—2 (Octal 010)
- Execute permission—1 (Octal 001)

For each category we add up the numbers. For instance, 6 represents read and write permissions, and 7 represents all permissions as can easily be understood from Table 6.2.

Table 6.2

Binary	*Octal*	*Permissions*	*Significance*
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely. The most significant digit represents user and the least one represents others. **chmod** can use this three-digit string as the expression.

You can use this method to assign read and write permission to all three categories. Without octal numbers, you should normally be using **chmod a+rw xstart** to achieve the task (assuming there was no execute permission initially). Now you can use a different method:

```
$ chmod 666 xstart ; ls -l xstart
-rw-rw-rw-   1 kumar    metal        1906 May 10 20:30 xstart
```

The 6 indicates read and write permissions (4 + 2). To restore the original permissions to the file, you need to remove the write permission (2) from group and others:

```
$ chmod 644 xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

Scanned by CamScanner

To assign all permissions to the owner, read and write permissions to the group, and only execute permission to the others, use this:

```
chmod 761 xstart
```

Now it should be quite easy for you to understand that the expression 777 signifies all permissions for all categories, while 000 indicates absence of all permissions for all categories. But can we delete a file with permissions 000? Yes, we can. Can we prevent a file with permissions 777 from being deleted? We can do that too. We'll soon learn that it's the directory that determines whether a file can be deleted, not the file itself.

Note: Remember that a file's permissions can only be changed by the owner (understood by **chmod** as user) of the file. One user can't change the protection modes of files belonging to another user. However, the system administrator can tamper with all file attributes including permissions, irrespective of their ownership.

6.5.3 The Security Implications

To understand the security implications behind these permissions and the role played by **chmod**, consider the default permissions of the file xstart that was shown at the beginning of Section 6.5:

```
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

These permissions are fairly safe; only the user can edit the file. What are the implications if we remove all permissions in either of these ways?

```
chmod u-rw,go-r xstart
chmod 000 xstart
```

The listing in either case will look like this:

```
----------   1 kumar    metal        1906 May 10 20:30 xstart
```

This setting renders the file virtually useless; you simply can't do anything useful with it. *But the user can still delete this file!* To understand why that can happen, you need to understand directory permissions and how they are related to file permissions.

On the other hand, you must not be too generous (and careless, too) to have all permissions enabled for all categories of users, using either of these commands:

```
chmod a+rwx xstart
chmod 777 xstart
```

The resulting permissions setting is simply dangerous:

```
-rwxrwxrwx   1 kumar    metal        1906 May 10 20:30 xstart
```

It's the universal write permission here that concerns us most. This file can be written by all. You shouldn't be able to read, write or execute every file. If that were possible, you can never have a secure system. The UNIX system, by default, never allows that, and no sensible user will compromise security so easily.

Note: We ignored the directory permissions in our discussions, but they also have a major role to play in setting a file's access rights. No matter how careful you are with your file permissions, a faulty directory permission will affect the security of all files housed in that directory. It doesn't matter who owns the file or whether the file itself has write permission for that user. Directory permissions are taken up later.

6.5.4 Using chmod Recursively (-R)

It's possible to make **chmod** descend a directory hierarchy and apply the expression to every file and subdirectory it finds. This is done with the -R (recursive) option:

```
chmod -R a+x shell_scripts
```

This makes all files and subdirectories found in the tree-walk (that commences from the shell_scripts directory) executable by all users. You can provide multiple directory and filenames, and if you want to use **chmod** on your home directory tree, then "cd" to it and use it in one of these ways:

```
chmod -R 755 .          Works on hidden files also
chmod -R a+x *          Leaves out hidden files
```

When you know the shell metacharacters well, you'll appreciate that the * doesn't match filenames beginning with a dot. The dot is generally a safer bet but note that both commands change the permissions of directories also. What do permissions mean when they are applied to a directory? The directory is taken up first in Section 6.6 and then again in Chapter 11.

6.6 DIRECTORY PERMISSIONS

Directories also have their own permissions and the significance of these permissions differ a great deal from those of ordinary files. You may not have expected this, but be aware that read and write access to an ordinary file are also influenced by the permissions of the directory housing them. It's possible that a file can't be accessed even though it has read permission, and can be removed even when it's write-protected. In fact, it's very easy to make it behave that way.

If the default directory permissions are not altered, the **chmod** theory still applies. However, if they are changed, unusual things can happen. Though directory permissions are taken up later *(11.4)*, it's worthwhile to know what the default permissions are on your system:

```
$ mkdir c_progs; ls -ld c_progs
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 c_progs
```

The default permissions of a directory on this system are rwxr-xr-x (or 755); that's what it should be. *A directory must never be writable by group and others.* If you find that your files are being tampered with even though they appear to be protected, check up the directory permissions. If the permissions differ from what you see here, look up Chapter 11 for remedial action.

Scanned by CamScanner

Caution: If a directory has write permission for group and others also, be assured that every user can remove every file in the directory! As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Note: The default file and directory permissions on your machine could be different from what has been assumed here. The defaults are determined by the *umask* setting of your shell. This topic is discussed in Section 11.5.

6.7 CHANGING FILE OWNERSHIP

File ownership is a feature often ignored by many users. By now you know well enough that when a user kumar of the metal group creates a file foo, he becomes the owner of foo, and metal becomes the group owner. It's only kumar who can change the major file attributes like its permissions and group ownership. No member of the metal group (except kumar) can change these attributes. However, when sharma copies foo, the ownership of the copy is transferred to sharma, and he can then manipulate the attributes of the copy at will.

There are two commands meant to change the ownership of a file or directory—**chown** and **chgrp**. UNIX systems differ in the way they restrict the usage of these two commands. On BSD-based systems, only the system administrator can change a file's owner with **chown**. On the same systems, the restrictions are less severe when it comes to changing groups with **chgrp**. On other systems, only the owner can change both.

6.7.1 chown: Changing File Owner

We'll first consider the behavior of BSD-based **chown** (change owner) that has been adopted by many systems including Solaris and Linux. The command is used in this way:

```
chown options owner [:group] file(s)
```

chown transfers ownership of a file to a user, and it seems that it can optionally change the group as well. The command requires the user-id (UID) of the recipient, followed by one or more filenames. Changing ownership requires superuser permission, so let's first change our status to that of superuser with the **su** command:

```
$ su
Password: ********                    This is the root password!
#                                     This is another shell
```

After the password is successfully entered, **su** returns a # prompt, the same prompt used by root. **su** lets us acquire superuser status if we know the root password. To now renounce the ownership of the file note to sharma, use **chown** in the following way:

```
# ls -l note
-rwxr----x   1 kumar    metal        347 May 10 20:30 note
# chown sharma note ; ls -l note
-rwxr----x   1 sharma   metal        347 May 10 20:30 note
# exit                                Switches from superuser's shell
$ _                                   to user's login shell
```


Once ownership of the file has been given away to sharma, the user file permissions that previously applied to kumar now apply to sharma. Thus, kumar can no longer edit note since there's no write privilege for group and others. He can't get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

6.7.2 chgrp: Changing Group Owner

By default, the group owner of a file is the group to which the owner belongs. The **chgrp** (change group) command changes a file's group owner. On systems that implement the BSD version of **chgrp** (like Solaris and Linux), a user can change the group owner of a file, but only to a group to which she also belongs. Yes, a user can belong to more than one group, and the one shown in /etc/passwd is the user's main group. We'll discuss *supplementary groups* in Chapter 15 featuring system administration.

chgrp shares a similar syntax with **chown**. In the following example, kumar changes the group ownership of dept.lst to dba (no superuser permission required):

```
$ ls -l dept.lst
-rw-r--r--   1 kumar    metal        139 Jun  8 16:43 dept.lst
$ chgrp dba dept.lst ; ls -l dept.lst
-rw-r--r--   1 kumar    dba          139 Jun  8 16:43 dept.lst
```

This command will work on a BSD-based system if kumar is also a member of the dba group. If he is not, then only the superuser can make the command work. Note that kumar can reverse this action and restore the previous group ownership (to metal) because he is still owner of the file and consequently retains all rights related to it.

Using chown *to Do Both* As an added benefit, UNIX allows the administrator to use only **chown** to change both owner and group. The syntax requires the two arguments to be separated by a :

```
chown sharma:dba dept.lst             Ownership to sharma, group to dba
```

Like **chmod**, both **chown** and **chgrp** use the -R option to perform their operations in a recursive manner.

chown Ownership:Group <file name>

6.8 CONCLUSION

In this chapter we considered two important file attributes—permissions and ownership. After we complete the first round of discussions related to files, we'll take up the other file attributes—the links and time stamps (Chapter 15). We also need to examine the inode that stores all of these attributes. In the next chapter, we'll discuss file editing with the **vi** editor.

WRAP UP

The **ls -l** command lists seven file attributes, and the output is often called the *listing*. **ls -ld** used with a directory name lists directory attributes.

Scanned by CamScanner

A file can have more than one link, i.e., more than one name without having multiple copies on disk.

A file has an owner, usually the name of the user (UID) who creates the file. A file is also owned by a group (GID), by default, the group to which the user belongs. Both UID and GID are maintained in /etc/passwd. Apart from the superuser, it's only the owner who can change all file attributes.

The file size displayed by the listing is not the actual space the file occupies on disk, but the number of bytes it contains.

A file can have read (r), write (w) or execute (x) permission, and there are three sets of such permissions for the user (u), *group* (g) and *others* (o).

A file's owner uses **chmod** to alter file permissions. The permissions can be *relative* when used with the + or - symbols, or *absolute* when used with octal numbers. The octal digit 7 includes read (4), write (2) and execute permission (1) bits.

For security reasons, a regular file shouldn't normally have write permission for group and others. A file's access rights are also influenced by the permissions of its directory which usually has read and execute permission for all.

chown and **chgrp** are used to transfer ownership and group ownership, respectively. They can be used by the owner of the file on AT&T systems. On BSD systems, **chown** can be used only by the superuser, and a user can use **chgrp** to change her group to another to which she also belongs.

Test Your Understanding

6.1 A file contains 1026 bytes. How many bytes of disk space will it occupy on a system where the size of a disk block is 1024 bytes?

6.2 Where are the UID and GID of a file stored?

6.3 Show the octal representation of these permissions: (i) rwxr-xrw- (ii) rw-r----- (iii) --x-w-r--

6.4 What will the permissions string look like for these octal values? (i) 567 (ii) 623 (iii) 421

6.5 If a file's permissions are 000, can the superuser still read and write the file?

6.6 Which important file attribute changes when you copy a file from another user's home directory?

6.7 Copy a file with permissions 444. Copy it again and explain your observations.

6.8 Try creating a directory in the system directories /bin and /tmp. What do you notice and why does that happen?

6.9 If the owner doesn't have write permission on a file but her group has, can she edit it?

6.10 How is **chown** different from **chgrp** on a BSD-based system when it comes to renouncing ownership?

6.11 Explain with reference to the dot and * what the following commands do: (i) **chown -R project .** (ii) **chgrp -R project ***

Flex Your Brain

6.1 How will you obtain a complete listing of all files and directories in the whole system and save the output in a file?

6.2 Explain briefly the significance of the first four fields of the **ls -l** output. Who can change these attributes? Is there any attribute that can be changed *only* by the superuser?

6.3 Explain the significance of the following commands: (i) **ls -ld .** (ii) **ls -l ..**

6.4 The commands **ls bar** and **ls -d bar** display the same output—the string bar. This can happen in two ways. Explain.

6.5 Does the owner always belong to the same group as the group owner of a file?

6.6 Create a file foo. How do you assign all permissions to the owner and remove all permissions from others using (i) relative assignment, (ii) absolute assignment? Do you need to make any assumptions about foo's default permissions?

6.7 You tried to copy a file foo from another user's directory, but you got the error message cannot create file foo. You have write permission in your own directory. What could be the reason and how do you copy the file?

6.8 Explain the consequences, from the security viewpoint, of a file having the permissions (i) 000 (ii) 777. Assume that the directory has write permission.

6.9 Examine the output of the two commands on a BSD-based system. Explain whether kumar can (i) edit, (ii) delete, (iii) change permissions, (iv) change ownership of foo:

```
$ who am i ; ls -l foo
kumar
-r--rw----    1 sumit    kumar          78 Jan 27 16:57 foo
```

6.10 Assuming that a file's current permissions are rw-r-xr--, specify the **chmod** expression required to change them to (i) rwxrwxrwx (ii) r--r----- (iii) --r--r-- (iv) --------, using both relative and absolute methods of assigning permissions.

6.11 Use **chmod -w .** and then try to create and remove a file in the current directory. Can you do that? Is the command the same as **chmod a-w foo**?

6.12 How will you determine whether your system uses the BSD or AT&T version of **chown** and **chgrp**?

Scanned by CamScanner

7

The vi Editor

No matter what work you do with the UNIX system, you'll eventually write some C programs or shell (or **perl**) scripts. You may have to edit some of the system files at times. For all this you must learn to use an editor, and UNIX provides a very versatile one—**vi**. Bill Joy created this editor for the BSD system. The program is now standard on all UNIX systems. Bram Moolenaar improved it and called it **vim** (vi improved). In this text we discuss the **vi** editor and also note the features of **vim** available in Linux.

Like any editor, **vi** uses a number of internal commands to navigate to any point in a text file and edit the text there. It also allows you to copy and move text within a file and also from one file to another. **vi** offers cryptic, and sometimes mnemonic, internal commands for editing work. It makes complete use of the keyboard where practically every key has a function. You don't need to master **vi** right now; a working knowledge is all that is required initially. The advanced features of **vi** are taken up in Part II of this book.

What You Will Learn

- The three modes in which **vi** operates for sharing the workload.
- Use a *repeat factor* to repeat a command multiple times.
- Use the *Input Mode* to insert and replace text.
- Use the *ex Mode* to save your work.
- Use the *Command Mode* to perform navigation.
- Use the *word* as a navigation unit for movement along a line.
- Delete, yank (copy) and move text using *operators*.
- Undo the last editing action and repeat the last command.
- Search for a pattern and repeat the search.
- Use the ex Mode to perform string substitution.

Topics of Special Interest

- Master the technique of using a three-function sequence to (i) search for a pattern (ii) take some action and (iii) repeat the search and action.

- Learn to use the two powerful features available in **vim**—word completion and multiple undoing.

7.1 vi BASICS

For a quick tour of **vi**, let's add some text to a file. Invoke **vi** with the filename **sometext**:

```
vi sometext
```

In all probability, the file doesn't exist, and **vi** presents you a full screen with the filename shown at the bottom with the qualifier, [New File]. The cursor is positioned at the top and all remaining lines of the screen (except the last) show a ~. You can't take your cursor there yet; they are nonexistent lines. The last line is reserved for commands that you can enter to act on text. This line is also used by the system to display messages.

You are now in the *Command Mode*, one of the three modes used by **vi**. This is the mode where you can pass commands to act on text, using most of the keys of the keyboard. Pressing a key doesn't show it on screen but may perform a function like moving the cursor to the next line, or deleting a line. You can't use the Command Mode to enter or replace text.

For text editing, **vi** uses 24 of the 25 lines that are normally available in a terminal. To enter text, you must switch to the *Input Mode*. First press the key marked i, and you are in this mode ready to input text. Subsequent key depressions will then show up on the screen as text input. Start inserting a few lines of text, each line followed by *[Enter]*, as shown in Fig. 7.1.

After text entry is complete, the cursor is positioned on the last character of the last line. This is known as the **current line** and the character where the cursor is stationed is the **current cursor position**. If you see something that shouldn't be there, use the backspace key to wipe it out. If a word has been misspelled, use *[Ctrl-w]* to erase the entire word.

Now press the *[Esc]* key to revert to Command Mode. Press it again and you'll hear a beep; a beep in **vi** indicates that a key has been pressed unnecessarily.

```
This is the vi editor[Enter]
It is slow in getting started but is quite powerful[Enter]
It operates in three modes[Enter]
All the features of ex are also available[Enter]
You can even escape to the UNIX shell[Enter]
It maintains 26 buffers for storing chunks of text
~
~
~
~
~
```

Fig. 7.1 Inserting Some Text

Scanned by CamScanner

The file sometext doesn't exist yet. Actually, the text that you entered hasn't been saved on disk but exists in some temporary storage called a *buffer*. To save the entered text, you must switch to the *ex Mode* (also known as *Last Line Mode*). Invoke the ex Mode from the Command Mode by entering a : (colon), which shows up in the last line. Enter an x and press *[Enter]*:

```
:x[Enter]                                    Quits editor—back to shell prompt
"sometext" 6 lines, 232 characters
$ _
```

The file is saved on disk and vi returns the shell prompt. To modify this file, you'll have to invoke **vi sometext** again. But before moving ahead, let's summarize the modes used by **vi**:

- **Command Mode**—The default mode of the editor where every key pressed is interpreted as a command to run on text. You'll have to be in this mode to copy and delete text. Unnecessary pressing of *[Esc]* in this mode sounds a beep but also confirms that you are in this mode.
- **Input Mode**—Every key pressed after switching to this mode actually shows up as text. This mode is invoked by pressing one of the keys shown in Table 7.1.
- **ex Mode** (or Last Line Mode)—The mode used to handle files (like saving) and perform substitution. Pressing a : in the Command Mode invokes this mode. You then enter an ex Mode command followed by *[Enter]*. After the command is run, you are back to the default Command Mode.

Much of the chapter deals with Command Mode commands where most of the action is. Some of these commands also have ex Mode equivalents which are sometimes easier to use. But all three modes also have their own exclusive features and an editing session in **vi** involves constant switching between modes as depicted in Fig. 7.2.

7.1.1 The Repeat Factor

When discussing **more** *(5.5)*, we introduced the *repeat factor* as a command prefix to repeat the command as many times as the prefix. **vi** also provides the repeat factor with many of its Command Mode and Input Mode commands. So if the Command Mode command **k** moves the cursor up one line, then **10k** moves it up 10 lines. The repeat factor thus speeds up operations. You'll be using it several times in this chapter.

7.1.2 The File .exrc

The default behavior of **vi** is adequate for novices, but as you get comfortable with it, you'll feel the need to customize it to behave in a way that makes writing programs and documents easier. **vi** reads the file **$HOME/.exrc** (same as **~/.exrc** in some shells) on startup. If **ls -a** doesn't show this file in your home directory, you can create or copy one. Linux users must note that **vim** generally doesn't use **.exrc** but **.vimrc** (with exceptions).

Many ex Mode commands can be placed in this file so they are available in every session. You can create abbreviations, redefine your keys to behave differently and also make variable settings. Your **.exrc** will progressively develop into an exclusive "library" containing all shortcuts and settings that you use regularly. It could be your most prized possession, so always keep a backup of this file.

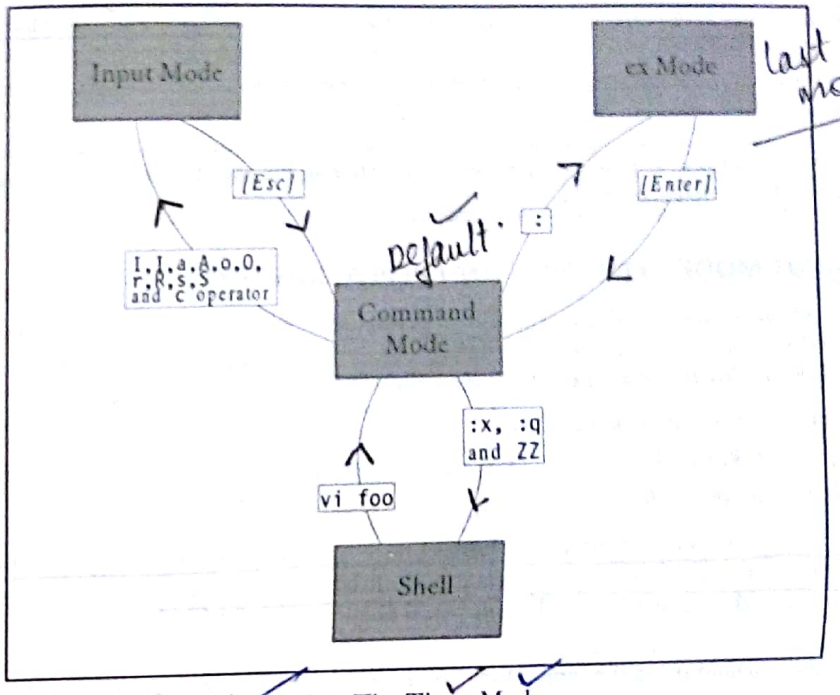


Fig. 7.2 The Three Modes

7.1.3 A Few Tips First

We are about to take off, but before we do that, a few tips at this stage will stand you in good stead. You must keep them in mind at all times when you are doing work with **vi**:

- *Undo whenever you make a mistake.* If you have made a mistake in editing, either by wrongly deleting text or inserting it at a wrong location, then as a first measure, just press *[Esc]* and then **u** to undo the last action. If that makes matters worse, use **u** again. Linux users should instead use *[Ctrl-r]*.
- *Clearing the screen* If the screen gets garbled for some reason, use *[Ctrl-l]* (el) in the Command Mode to redraw the screen. If you hit *[Ctrl-l]* in the Input Mode, you'll see the symbol ^L on the screen. Use the backspace key to wipe it out, press *[Esc]* and then hit *[Ctrl-l]*.
- *Don't use [CapsLock]* **vi** commands are case-sensitive; **a** and **A** are different commands. Even if you activate this key to enter a large block of text in uppercase, make sure you deactivate it after text entry is complete.
- *Avoid using the PC navigation keys* As far as possible, avoid using all the standard navigation keys like Up, Down, Left and Right, *[PageUp]* and *[PageDown]*. Many of them could fail when you use **vi** over a network connection. **vi** provides an elaborate set of keys for navigation purposes.

Scanned by CamScanner

- vi reads the TERM variable to determine the file that contains the terminal's characteristics. As discussed later (10.3), vi actually reads a system file to know the control sequences that apply to the terminal name assigned to TERM. You should always check TERM whenever vi behaves in an awkward manner.

Note: Only the keys g, K, q, v, V and Z have no function in the standard vi implementation. Some of them are defined, however, in vim.

7.2 INPUT MODE—ENTERING AND REPLACING TEXT

In this section, we take up all the commands that let you enter the Input Mode from the Command Mode. When a key of the Input Mode is pressed, it doesn't appear on screen but subsequent key depressions do. We'll consider the following commands:

- Insert and append (i, a, I and A)
- Replace (r, R, s and S)
- Open a line (o and O)

Always keep in mind that after you have completed text entry using any of these commands (except r), you must return to the Command Mode by pressing *[Esc]*. Most of these commands can also be used with a repeat factor, though you'll need to use it with only some of them.

Tip: Before you start using the Input Mode commands, enter this ex Mode command:

```
:set showmode[Enter]
```

Enter a : (the ex Mode prompt) and you'll see it appear in the last line. Follow it with the two words and press *[Enter]*. showmode sets one of the parameters of the vi environment. Messages like INSERT MODE, REPLACE MODE or CHANGE MODE, etc. will now appear in the last line when you run an Input Mode command. We'll learn later to make the setting permanent by placing it in $HOME/.exrc.

7.2.1 Insertion of Text (i and a)

The simplest type of input is insertion of text. Just press

```
i
```

Existing text will be shifted right

Pressing this key changes the mode from Command to Input. Since the showmode setting was made at the beginning (with :set showmode), you'll see the words INSERT MODE at the bottom-right corner of the screen. Further key depressions will result in text being entered and displayed on the screen.

If the i command is invoked with the cursor positioned on existing text, text on its right will be shifted further without being overwritten. The insertion of text with i is shown in Fig. 7.3, along with the position of the cursor.


```
the vi editor
   ↓ ifull-screen [Esc]
the vi full-screen editor
```

Fig. 7.3 Text Insertion with i

```
the vi full-screen editor
   ↓ a, a link of ex[Esc]
the vi full-screen editor, a link of ex
```

Fig. 7.4 Text Appending with a

There are other methods of inputting text. To append text to the right of the cursor position, use

```
a
```

Existing text will also be shifted right

followed by the text you wish to key in (Fig. 7.4). After you have finished editing, press *[Esc]*. With i and a, you can append several lines of text in this way.

7.2.2 Insertion of Text at Line Extremes (I and A)

I and A behave somewhat like i and a except that they work at line extremes by performing the necessary navigation to move there:

- I Inserts text at beginning of line.
- A Appends text at end of line.

These two commands are suitable for converting code to comment lines in a C program (Fig. 7.5). A comment line in C is of the form /* *comment* */. Use I on an existing line that you now wish to convert to a comment, and then enter the symbols /*. After pressing *[Esc]*, use A to append */ at the end of the line and press *[Esc]* again. A document author often needs to use A to continue text entry from the point work was suspended—adding a sentence to a paragraph, for instance.

```
The following code forks a process
   ↓ I/* [Esc]
/*The following code forks a process
   ↓ A */[Esc]
/* The following code forks a process */
```

Fig. 7.5 Using I and A to Create a Comment Line in a C Program

Scanned by CamScanner

```
vi and ex are one and the same editor
oIt is due to William Joy[Esc]
vi and ex are one and the same editor
It is due to William Joy
```

Fig. 7.6 Opening a New Line with **o**

Tip: C language programmers sometimes precede a block of comment lines with a pattern like this:

```
/**************************************************************
```

It seems that there are 70 asterisks here, and to draw them you should use a repeat factor. After you have entered / in the Input Mode, press *[Esc]*, and then enter **70a*** *[Esc]*. You'll see 70 asterisks appended to the /.

7.2.3 Opening a New Line (o and O)

Many a time it has been observed that people migrating from a word-processing environment use a convoluted sequence to open a new line below the current line. They move the cursor past the end of the current line and press *[Enter]*. In **vi**, you use **o** and **O** instead. To open a line below or above anywhere in a line, simply press

```
o                    Opens new line below current line
```

This inserts an empty line below the current line (Fig. 7.6). **O** also opens a line but above the current line. You are free to enter as much text as you choose, spanning multiple lines if required. Press the *[Esc]* key after completing text input.

CAPITAL "O"

7.2.4 Replacing Text (r, s, R and S)

To change existing text, **vi** provides mainly four commands (actually, more) as shown in the next few sections. To replace a single character with another, you should use

```
r                    No [Esc] required
```

Replace one character with one.

```
This is the vi full-screen editor
rT
This is the vi full-screen editor
```

Fig. 7.7 Replacing a Single Character with **r**

Replace one character with many.

```
vi is a link of ex
sone[Esc]
vi is one link of ex
```

Fig. 7.8 Replacing Text with **s**

```
vi is one of the links of the ex program
Rmodes[Esc]
vi is one of the modes of the ex program
```

Fig. 7.9 Replacing Text with **R**

followed by the character that replaces the one under the cursor (Fig. 7.7). You can replace a single character only in this way. **vi** momentarily switches from Command Mode to Input Mode when **r** is pressed. It returns to Command Mode as soon as the new character is entered. There's no need to press *[Esc]* when using **r** and the replacement character, since **vi** expects a single character anyway.

When you want to replace the letter d with 10f in a **printf** statement in C, you need to replace one character with three. In that case, press

```
s                    Replaces one character with many
```

vi deletes the character under the cursor and switches to Input Mode. It may also show a **$** at that location to indicate that replacement will not affect text on its right. Now enter **10f** and press *[Esc]*. To replace multiple characters, use a repeat factor. **3s** replaces three characters with new text. Use of **s** is shown in Fig. 7.8.

R and **S** act in a similar manner compared to their lowercase ones except that they act on a larger group of characters:

R	Replaces all text on the right of the cursor position.
S	Replaces the entire line irrespective of cursor position. (Existing line disappears)

Try using the repeat factor with **R** and **S** and see whether you can take advantage of this feature. Use of **R** is shown in Fig. 7.9.

You have now been able to enter the Input Mode in ten ways. The functions of these ten keys are summarized in Table 7.1.

Caution: You must press *[Esc]* to switch to Command Mode after you have keyed in text. Repeated pressing of *[Esc]* won't make any difference to **vi** except that it has a built-in capability to indicate with a beep if a key has been pressed unnecessarily. Try this by pressing *[Esc]* several times. You are now in the Command Mode.

LINUX: A superb text completion feature is available in **vim**. If the string **printf** is available in the file, you don't need to enter the entire string eve. Just key in as much as is necessary to make the string unique (say, up to pr), and then press

```
[Ctrl-p]             vim attempts to complete string
```

Scanned by CamScanner

vim expands pr to printf if this is the *only* word beginning with pr. In case there are other words, repeated pressing of the key shows all matching words in turn. In case you have to view the list backwards, use *[Ctrl-n]*.

Table 7.1 Input Mode Commands

Command	Function
i	Inserts text to left of cursor (Existing text shifted right)
a	Appends text to right of cursor (Existing text shifted right)
I	Inserts text at beginning of line (Existing text shifted right)
A	Appends text at end of line
o	Opens line below
O	Opens line above
rch	Replaces single character under cursor with *ch* (No *[Esc]* required)
R	Replaces text from cursor to right (Existing text overwritten)
s	Replaces single character under cursor with any number of characters
S	Replaces entire line

7.3 SAVING TEXT AND QUITTING—THE ex MODE

When you edit a file using vi—or for that matter, any editor—the original file isn't disturbed as such, but only a copy of it that is placed in a **buffer** (a temporary form of storage). From time to time, you should save your work by writing the buffer contents to disk to keep the disk file current (or, as we say, in *sync*). When we talk of saving a file, we actually mean saving this buffer. You may also need to quit vi after or without saving the buffer. These features are adequately handled by the ex Mode. The basic file handling features are shown in Table 7.2.

7.3.1 Saving Your Work (:w)

We have already used the ex Mode command, :x, to save the buffer and exit the editor *(7.1)*. For extended sessions with vi, you must able to save the buffer and *remain* in the editor. From time to time, you must use the :w command to write the buffer to disk. Enter a :, which appears on the last line of the screen, then w and finally *[Enter]*:

```
:w[Enter]
"sometext", 8 lines, 275 characters
```

You can now continue your editing work normally, but make sure that you execute this command regularly. With the :w command you can optionally specify a filename as well. In that case, the contents are separately written to another file.

Tip: It's common practice to ignore the `readonly` label on opening a file that doesn't have the write permission bit set. When you attempt to save the file with :w, vi retorts with the message `File is read only`. You should have been careful in the first place, but there's hope: Just save the file with a different name (say, `:w foo`) after making sure that `foo` doesn't exist. Look up Table 7.2 for the command to use when `foo` exists.

7.3.2 Saving and Quitting (:x and :wq)

The previous command returns you to the Command Mode so you can continue editing. However, to save and quit the editor (i.e., return to the shell), use the :x (exit) command instead:

```
:x[Enter]
"sometext", 8 lines, 303 characters
$ _
```

You can also use :wq to save and quit the editor. But that requires an additional keystroke and is not recommended for use.

Tip: The best way to save and quit the editor is to use ZZ, a Command Mode command, instead of :x or :wq.

7.3.3 Aborting Editing (:q)

It's also possible to abort the editing process and quit the editing mode without saving the buffer. The :q (quit) command does that job:

```
:q[Enter]
$ _
```

vi also has a safety mechanism that prevents you from aborting accidentally if you have modified the file (buffer) in any way. The following message is typical when you try to do so:

```
No write since last change (:quit! overrides)
```

If the buffer has been changed and you still want to abandon the changes, then use

```
:q!
```

to return you to the prompt irrespective of the status of the buffer—no questions asked. vi suggests appending a ! to an ex Mode command every time it feels that you could be doing something that is potentially unsafe.

Note: In general, any ex Mode command used with a ! signifies an abort of some type. It can be used to switch to another file without saving the current one, or reload the last saved version of a file. You can even use it to overwrite a separate file.

7.3.4 Writing Selected Lines

The :w command is an abbreviated way of executing the ex Mode instruction :1,$w. w can be prefixed by one or two *addresses* separated by a comma. The command

```
:10,50w n2words.pl
```

saves lines 10 through 50 to the file `n2words.pl`. You can save a single line as well:

```
:5w n2words.pl
```

Scanned by CamScanner

There are two symbols used with w that have special significance—the dot and $. The dot represents the current line while $ represents the last line of the file. You can use them singly or in combination:

```
:.w tempfile      Saves current line (where cursor is positioned)
:$w tempfile      Saves last line
:.,$w tempfile    Saves current line through end
```

If tempfile exists and is writable by you, vi issues yet another warning:

```
"tempfile" File exists - use "w! tempfile" to overwrite
```

The message is clear: vi is telling you to suffix the ! to the :w command to overwrite tempfile. As mentioned before, the ! is the universal overriding operator in the ex Mode and you'll be using it often.

Note: In the ex Mode, the current line number is represented by . (dot) and the last line is denoted by $. :w is the same as :1,$w. Both commands address the entire file.

7.3.5 Escape to the UNIX Shell (:sh and *[Ctrl-z]*)

How do do you edit and compile your C program repeatedly? You need to make a temporary escape to the shell to run the cc command. There are two ways; the first method is to use the ex Mode command, :sh:

```
:sh
$ _                    You haven't quit vi yet
```

This returns a shell prompt. Execute cc or any UNIX command here and then return to the editor using *[Ctrl-d]* or exit. Don't make the mistake of running vi once again, as you'll then have two instances of vi—an undesirable situation.

If your shell supports job control (which most shells do), you can also *suspend* the current vi session. Just press *[Ctrl-z]* and you'll be returned a shell prompt. Run your commands and then use the fg command to return to the editor. Job control is discussed in Section 9.10.

7.3.6 Recovering from a Crash (:recover and -r)

Accidents can and will happen. The power can go off, leaving work unsaved. However, don't panic, vi stores most of its buffer information in a hidden swap file. Even though vi removes this file on successful exit, a power glitch or an improper shutdown procedure lets this swap file remain on disk. vi will then complain the next time you invoke it with the same file.

The complaint usually also contains some advice regarding the salvage operation. You'll be advised to use either the ex Mode command :recover, or vi -r foo to recover as much of foo as possible. After you have done that, have a look at the buffer contents and satisfy yourself of the success of the damage control exercise. If everything seems fine, save the buffer and remove the swap file if vi doesn't do that on its own.

Note: You can't be assured of complete recovery every time. Sometimes, vi may show you absolute junk when using the -r option (or :recover). In that case, don't save the file and simply quit (with :q!). Start vi again normally; recovery is not possible here. Linux users should note that in these situations, they need to delete the file having a .swp extension manually; otherwise the file will not be editable.

Table 7.2 Save and Exit Commands of the ex Mode

Command	*Action*
:w	Saves file and remains in editing mode
:x	Saves file and quits editing mode
:wq	As above
:w n2w.pl	Like *Save As* ..... in Microsoft Windows
:w! n2w.pl	As above, but overwrites existing file
:q	Quits editing mode when no changes are made to file
:q!	Quits editing mode but after abandoning changes
:*n1*,*n2*w build.sql	Writes lines *n1* to *n2* to file build.sql
:.w build.sql	Writes current line to file build.sql
:$w build.sql	Writes last line to file build.sql
:!*cmd*	Runs *cmd* command and returns to Command Mode
:sh	Escapes to UNIX shell
:recover	Recovers file from a crash

7.4 NAVIGATION

We'll now consider the functions of the Command Mode. This is the mode you come to when you have finished entering or changing your text. A Command Mode command doesn't show up on screen but simply performs a function. We begin with navigation. Don't forget to avoid the cursor control keys for navigation as advised in Section 7.1.3.

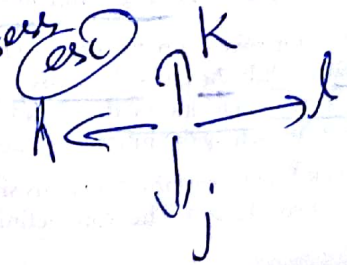
7.4.1 Movement in the Four Directions (h, j, k and l)

vi provides the keys h, j, k and l to move the cursor in the four directions. These keys are placed adjacent to one another in the middle row of the keyboard. Without a repeat factor, they move the cursor by one position. Use these keys for moving the cursor vertically:

k	Moves cursor up
j	Moves cursor down

To move the cursor along a line, use these commands:

h	Moves cursor left
l	Moves cursor right



Scanned by CamScanner

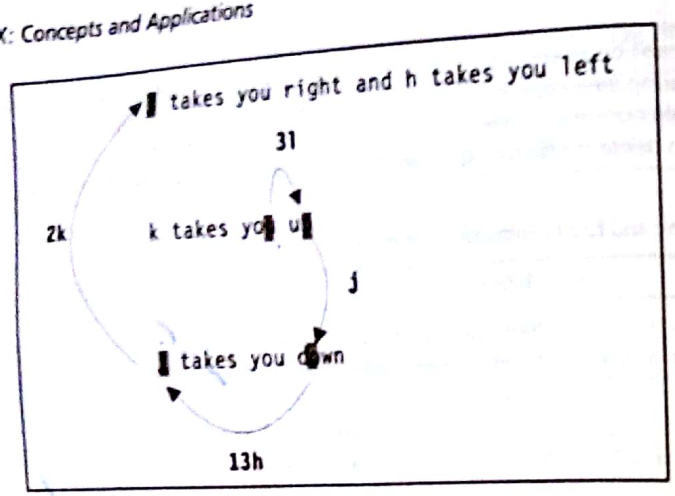


Fig. 7.10 Relative Navigation with h, j, k and l

The repeat factor can be used as a command prefix with all these four commands. Thus, **4k** moves the cursor 4 lines up and **20h** takes it 20 characters to the left. Navigation with the four keys is shown in Fig. 7.10. Note that this motion is relative; you can't move to a specific line number with these keys.

Tip: To remember the keys that move the cursor left or right, observe these four keys on your keyboard. The left-most key, **h**, moves the cursor to the left, and the right-most key, **l** (el), moves it right.

7.4.2 Word Navigation (b, e and w)

Moving by one character is not always enough; you'll often need to move faster along a line. **vi** understands a *word* as a navigation unit which can be defined in two ways, depending on the key pressed. If your cursor is a number of words away from your desired position, you can use the word-navigation commands to go there directly. There are three basic commands:

- **b** Moves back to beginning of word
- **e** Moves forward to end of word
- **w** Moves forward to beginning of word

A repeat factor speeds up cursor movement along a line. For example, **5b** takes the cursor five words back, while **3w** takes the cursor three words forward. A word here is simply a string of alphanumeric characters and the _ (underscore). `Bash` is one word; so is `sh_profile`. `tcp-ip` is three words; the hyphen by definition becomes a word too.

The keys **B**, **E** and **W** perform functions similar to those of their lowercase counterparts except that punctuation is skipped. The word definition also gets changed here, but we'll ignore these minor details.

7.4.3 Moving to Line Extremes (0, | and $)

Moving to the beginning or end of a line is a common requirement. This is handled by the keys **0**, **|** and **$**. To move to the first character of a line, use

0 *(zero) or* **|** — **30|** *moves cursor to column 30*

The **|** takes a repeat factor and using that, you can position the cursor on a certain column. To position the cursor on column 30, use **30|**.

We used **$** as the line address in the ex Mode to represent the last line of the file. The same symbol in the Command Mode represents the end of line. To move to the end of the current line, use

$ — *Moves to end of line*

The use of these two commands along with those that use units of words (**b**, **e** and **w**) is shown in Fig. 7.11.

Note: When you use the keys **b**, **e** and **w**, remember that a word is simply a string of alphanumeric characters and the _ (underscore).

7.4.4 Scrolling ([Ctrl-f], [Ctrl-b], [Ctrl-d] and [Ctrl-u])

Faster movement can be achieved by scrolling text in the window using the control keys. The two commands for scrolling a page at a time are

- *[Ctrl-f]* Scrolls forward
- *[Ctrl-b]* Scrolls backward

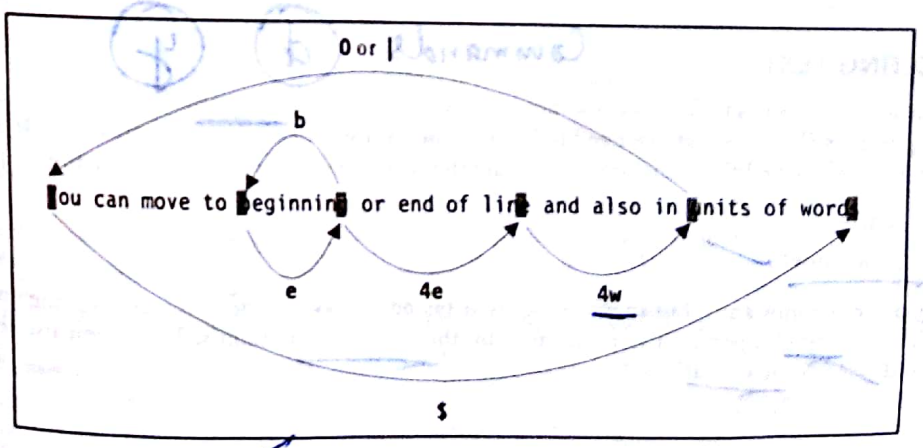


Fig. 7.11 Navigation Along a Line

Scanned by CamScanner

You can use the repeat factor, like in **10***[Ctrl-f]*, to scroll 10 pages and navigate faster in the process. You can scroll by half a page as well:

[Ctrl-d]	Scrolls half page forward
[Ctrl-u]	Scrolls half page backward

The repeat factor can also be used here.

7.4.5 Absolute Movement (G)

Upon startup, **vi** displays the total number of lines in the last line. At any time, you can press *[Ctrl-g]* to know the current line number:

```
"/etc/passwd" [Read only] line 89 of 179 --49%--
```

The cursor is on line 89 (49% of 179), and this read-protected file has 179 lines in all. Compilers also indicate line numbers in their error messages. You need to use the **G** command with the line number as repeat factor to locate the offending lines. To move to the 40th line, use

```
40G                    Goes to line number 40
```

and to move to the beginning of the file, use

```
1G                     Goes to line number 1
```

The end of the file is reached by simply using

```
G                      Goes to end of file
```

Note: The ex Mode offers equivalent commands for moving between lines. The previous three commands can be replaced with **:40**, **:1** and **:$**, respectively (along with *[Enter]*).

7.5 EDITING TEXT

The editing facilities in **vi** are very elaborate and involve the use of *operators*. Though operators are taken up in Part II of this text, we need to know a special form of their usage now to enable us to perform the essential editing functions. These are the two operators that we will use in this chapter:

d	Delete
y	Yank (copy)

d and **y** are not commands, but they can be used (as **dd** and **yy**) for deleting and copying entire lines. The "pasting" operation is performed by the **p** and **P** commands. We'll soon use these commands for editing operations.

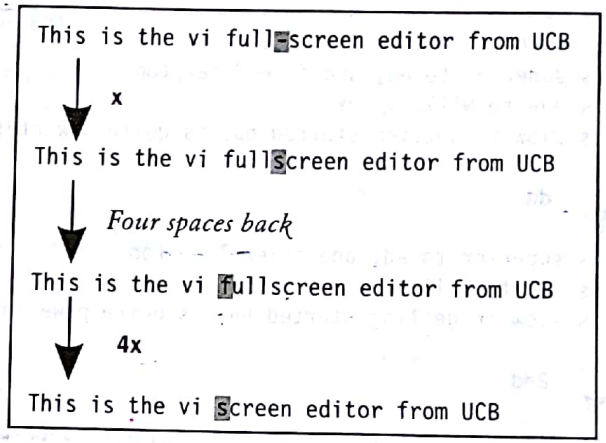


Fig. 7.12 Deleting Text with **x**

7.5.1 Deleting Text (x and dd)

The simplest text deletion is achieved with the **x** command. This command deletes the character under the cursor. Move the cursor to the character that needs to be deleted and then press

```
x                      Deletes a single character
```

The character under the cursor gets deleted, and the text on the right shifts left to fill up the space. A repeat factor also applies here, so **4x** deletes the current character as well as three characters from the *right* (Fig. 7.12).

A Windows Notepad user would be surprised to note that when the cursor is at the end of a line, **x** doesn't pull up the following line but works instead on text on the *left* of the cursor. Deletion of text from the left is otherwise handled by the **X** command. Keep it pressed, and you'll see that you have erased all text to the beginning of the line.

Entire lines are removed with the **dd** "command" (rather a doubled operator). Move the cursor to any line and then press

```
dd                     Cursor can be anywhere in line
```

6dd deletes the current line and five lines below. Fig. 7.13 illustrates the use of **dd** both with and without a repeat factor. There are other forms of deletion available in **vi** and you'll know them all after you have understood the **d** operator well *(20.1.1)*.

7.5.2 Moving Text (p, P)

Text movement requires you to perform an additional task: Put the text at the new location with **p** or **P**. **vi** uses these two commands for all "put" operations that follow delete or copy operations. The significance of **p** and **P** depends on whether they are used on parts of lines or complete lines. We need some examples to illustrate their behavior.

Scanned by CamScanner

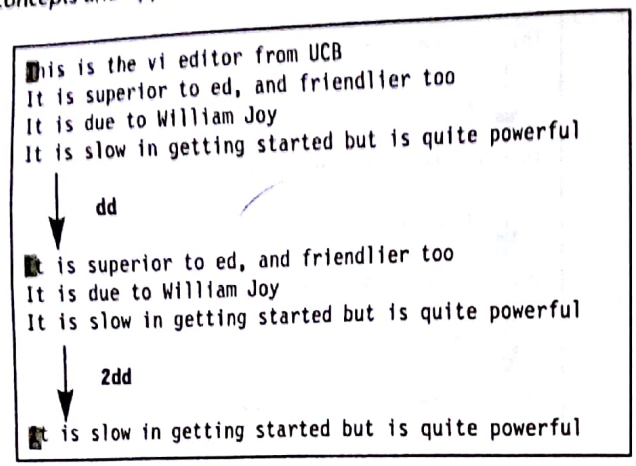


Fig. 7.13 Deleting Lines with **dd**

For instance, to correct `sdtio.h` to `stdio.h`, you'll have to transpose the characters **d** and **t**, i.e. delete the **d** and put it after the **t**. Move your cursor to the **d** in `sdtio.h` and use these two commands:

x	`sdtio.h` *becomes* `stio.h`—*cursor on t*
p	**d** *put on right*—`stio.h` *becomes* `stdio.h`

Since we worked on parts of lines, associate the word "right" with **p** and "left" with **P**. **P** places text on the left of the cursor.

To put entire (deleted) lines at a different location, you'll need to use the same commands, except that we have to think in terms of "below" and "above" the current line. **p** places text below the current line and **P** places text above.

Note: **p** and **P** place text on right and left only when you delete parts of lines. But the same keys get associated with "below" and "above" when you delete complete lines.

7.5.3 Copying Text (y and p)

vi uses the term **yanking** for copying text, the reason why the operator is named **y**. The principles are exactly the same as compared to deletion with **d**. For instance, to copy (or yank) one or more lines, use the **yy** "command":

yy	*Yanks current line*
10yy	*Yanks current line and 9 lines below*

This yanked text has to be placed at the new location. The put commands are the same—**p** and **P**. Since we copied entire lines, we can only place the copied text below or above the current line, and not on the left or right. But suppose you need to place an entire line at the end of another line. You have to place it below the line first and then join the two lines. Just read on.

7.5.4 Joining Lines (J)

In word processors, you join the current and next line by moving the cursor to the end of line and pressing *[Delete]*. This technique won't work in **vi**. To join the current line and the line following it, use

J	**4J** *joins following 3 lines with current one*

J removes the newline character between the two lines to pull up the line below it. Joining, however, is restricted to the maximum line size that your **vi** editor can support. It could be below 2000 (as in Solaris) or unlimited (in Linux).

To solve the problem posed at the end of the previous section, if you place a line below the current line with **p**, you can use **J** to join the two lines.

7.6 UNDOING LAST EDITING INSTRUCTIONS (u and U)

If you have inadvertently made an editing change that you shouldn't have, **vi** provides the **u** command to undo the last change made. Before you do anything else, reverse the last change you made to the buffer by pressing

u	*Must use in Command Mode; press [Esc] if necessary*

This will undo the most recent single editing change by restoring the position before the change. Another **u** at this stage will undo this too, i.e., restore the original status (doesn't apply to Linux). This facility is very useful, especially for beginners, who may accidentally delete a group of lines. Any deletion can be undone with **u** provided it is pressed *immediately* after deletion, and before fresh editing action has been performed.

When a number of editing changes have been made to a single line, **vi** allows you to discard all changes *before you move away from the line*. The command

U	*Don't move away from current line*

reverses all changes made to the current line, i.e., all modifications that have been made since the cursor was moved to this line.

Caution: Make sure the cursor has not been moved to another line before invoking **U**, in which case it won't work.

LINUX: *Multiple Undoing and Redoing* **vim** lets you undo and redo multiple editing instructions. **u** behaves differently here; repeated use of this key progressively undoes your previous actions. You could even have the original file in front of you! Further, **10u** reverses your last 10 editing actions. The function of **U** remains the same.

You may overshoot the desired mark when you keep **u** pressed, in which case use *[Ctrl-r]* to redo your undone actions. Further, undoing with **10u** can be completely reversed with **10***[Ctrl-r]*. The undoing limit is set by the ex Mode command **:set undolevels**=*n*, where *n* is set to 1000 by default.

Scanned by CamScanner

7.7 REPEATING THE LAST COMMAND (.)

Most editors don't have the facility to repeat the last *editing* instruction, but **vi** has. The **.** (dot) command is used for repeating both Input and Command Mode commands that perform editing tasks. The principle is this: Use the actual command only once, and then repeat it at other places with the dot command.

To take a simple example, if you have deleted two lines of text with **2dd**, then to repeat this operation elsewhere, all you have to do is to position the cursor at the desired location and press

Use **u** *to undo this repeat*

.

This will repeat the last editing instruction performed, i.e., it will also delete two lines of text.

The **.** command is indeed a very handy tool. As another example, consider that you have to indent a group of lines by inserting a tab at the beginning of each line. You need to use **i***[Tab][Esc]* only once, say on the first line. You can then move to each line in turn by hitting *[Enter]*, and simply press the dot. A group of lines can be indented in no time. The dot is specially suited for substitution work also, as you'll see in Section 7.8.1.

The **.** command can be used to repeat only the most recent *editing* operation— be it insertion, deletion or any other action that modifies the buffer. This, obviously, doesn't include the search commands (that don't modify the buffer contents in any way) which have their own set of characters for repeating a search. It doesn't also include the navigation or paging commands because they don't alter the buffer either.

Note: The dot command can be applied to repeat the last editing instruction only, i.e., an instruction that changes the contents of a file. The dot retains its meaning even after undoing, so you can use it again at another location.

7.8 SEARCHING FOR A PATTERN (/ and ?)

vi is extremely strong in search and replacement activities. Searching can be made in both forward and reverse directions, and can be repeated. It is initiated from the Command Mode by pressing a **/**, which shows up in the last line. For example, if you are looking for the string `printf`, enter this string after the /:

/`printf`*[Enter]*

Searches forward

The search begins forward to position the cursor on the first instance of the word. **vi** searches the entire file, so if the pattern can't be located until the end of file is reached, the search wraps around to resume from the beginning of the file. If the search still fails, **vi** responds with the message `Pattern not found.`

Likewise, the sequence

?pattern[Enter]

searches backward for the most previous instance of the pattern. The wraparound feature also applies here but in the reverse manner.

7.8.1 Repeating the Last Pattern Search (n and N)

The **n** and **N** commands repeat a search where **n** and **N** don't exactly play the roles you'd expect them to. For repeating a search in the direction the previous search was made with **/** or **?**, use

n *Repeats search in same direction of original search*

The cursor will be positioned at the beginning of the pattern. In this way, you can press **n** repeatedly to scan all instances of the string. **N** reverses the direction pursued by **n**, which means that you can use it to retrace your search path. The search and repeat actions are illustrated in Fig. 7.14 and the commands are summarized in Table 7.3.

Note: **n** doesn't necessarily repeat a search in the forward direction; the direction depends on the search command used. If you used **?printf** to search in the reverse direction in the first place, then **n** also follows the same direction. In that case, **N** will repeat the search in the forward direction, and not **n**.

Tip: The three commands, **/** (search), **n** (repeat search) and **.** (repeat last editing command), form a wonderful trio of search—search-repeat—edit-repeat commands. You'll often be tempted to use this trio in many situations where you want the same change to be carried out at a number of places.

For instance, if you want to replace some occurrences of `int` with `double`, then first search for `int` with **/int**, change `int` to `double` with **3s**, repeat the search with **n**, and press the **.** wherever you want the replacement to take place. Yes, you wouldn't like `printf` to also show up (`int` is embedded there), which means you need to use a *regular expression* to throw `printf` out. Like **more**, **vi** also recognizes regular expressions as search patterns; these expressions are first discussed in Section 13.2.

```
/echo [Enter]   if [ -z "$pname" ]
                then
                    echo "You have not entered the string"
                    exit 1
            n   else
                    echo "Enter the file to be used: \c"
            n       read flname
                    if [ ! -n "$flname" ] ; then
            N           echo "You have not entered the filename" ; exit 2
                    else
```

Fig. 7.14 Search and Repeat with / and **n**

Scanned by CamScanner

Note: You can start up vi by specifying a pattern. Use the +/ symbols before the pattern:

```
vi +/scheduling chap01
```

Cursor at scheduling

The cursor will be located at the first instance of the pattern. You can then use n and N in the usual for locating the next instance of the string. If the pattern contains multiple words, enclose within quotes.

Table 7.3 Search and Repeat Commands

Command	*Function*
/*pat*	Searches forward for pattern *pat*
?*pat*	Searches backward for pattern *pat*
n	Repeats search in same direction along which previous search was made
N	Repeats search in direction opposite to that along which previous search was

7.9 SUBSTITUTION—SEARCH AND REPLACE (:s)

vi offers yet another powerful feature, that of substitution, which is achieved with the ex M s (substitute) command. It lets you replace a pattern in the file with something else. The syntax shown below:

```
:address/source_pattern/target_pattern/flags
```

The *source_pattern* here is replaced with *target_pattern* in all lines specified by *address*. The *add* can be one or a pair of numbers, separated by a comma. For instance, 1,$ addresses all lines file. The most commonly used flag is g, which carries out the substitution for all occurrences of pattern in a line. The following example shows a typical substitution command:

```
:1,$s/director/member/g
```

Can also use % instead of 1,$

Here, director is replaced with member globally throughout the file. If the pattern can't be fou vi responds with the message Substitute pattern match failed. If you leave out the g substitution will be carried out for the *first occurrence in each* addressed line.

The target pattern is optional. If you leave it out, then you'll delete all instances of the sou pattern in all lines matched by the address:

```
:1,50s/unsigned//g
```

Deletes unsigned *everywhere in lines 1 to 50*

As shown above, you can choose the range of lines that are to be affected by the substitution. T following examples should make addressing clear:

```
:3,10s/director/member/g
:.s/director/member/g
:$s/director/member/g
```

Substitute lines 3 through 10
Only the current line
Only the last line

Interactive Substitution Sometimes you may like to selectively replace a string. In that case, add the c (confirmatory) parameter as the flag at the end:

```
:1,$s/director/member/gc
```

Each line is selected in turn, followed by a sequence of carets in the next line, just below the pattern that requires substitution. The cursor is positioned at the end of this caret sequence, waiting for your response:

```
9876|jai sharma       |director |production|03/12/50|7000
                              ^^^^^^^y
2365|barun sengupta   |director |personnel |05/11/47|7800
                              ^^^^^^^n
1006|chanchal singhvi |director |sales     |09/03/38|6700
                              ^^^^^^^y
6521|lalit chowdury   |director |marketing |09/26/45|8200
                              ^^^^^^^n
```

A y performs the substitution, any other response doesn't. This sequence is repeated for each of the matched lines in turn. In the present case, the substitution is performed for only two of the four lines.

LINUX: The interactive substitution feature in **vim** is both friendlier and more powerful than its UNIX counterpart. The string to be replaced is shown in reverse video, and a prompt appears in the last line of the screen:

```
replace with msg (y/n/a/q/^E/^Y)?
```

Apart from responding with y or n, you have the option of aborting (q) the substitution process or making it noninteractive (a).

7.10 CONCLUSION

The features of vi that have been highlighted so far are good enough for a beginner who shouldn't proceed any further before mastering most of them. There are many more functions that make vi a very powerful editor. Can you copy three words or even the entire file using simple keystrokes? Can you copy or move multiple sections of text from one file to another in a single file switch? How do you compile your C and Java programs without leaving the editor? vi can do all that, and if you already know this editor quite well, then skip to Chapter 20.

With the completion of the first round of discussions on files, we need to examine the other areas of UNIX. The shell is taken up in the next chapter.

WRAP UP

vi operates in three modes. The Command Mode is used to enter commands that operate on text or control cursor motion. The Input Mode is used to enter text. The ex Mode (or Last Line Mode) is used for file handling and substitution.

Scanned by CamScanner

Most of the Input and Command Mode commands also work with a *repeat factor* which performs the command multiple times.

The Input Mode is used to insert (**i** and **I**), append (**a** and **A**), replace (**r** and **R**), change (**s** or **S**) text and open a line (**o** and **O**). The mode is terminated by pressing *[Esc]*.

Using the ex Mode you can save your work (**:w**), exit the editor after saving (**:x**) and quit without saving (**:q** and **:q!**). The current line in this mode is represented by a dot and the last line by a **$**.

Navigation is performed in the Command Mode. You can move in the four directions (**h**, **j**, **k** and **l**) or move along a line, using a word as a navigation unit. You can move back (**b**) and forward (**w**) to the beginning of a word.

Editing functions are also performed in the Command Mode. You can delete characters (**x** and **X**) and lines (**dd**), and yank or copy lines (**yy**). Deleted and yanked text can be put at another location (**p** and **P**).

vi can repeat (**.**) and undo (**u**) the last editing instruction. **vim** in Linux can perform multiple levels of undo and redo with **u** and *[Ctrl-r]*, respectively.

You can search for a pattern (**/** and **?**) and also repeat (**n** and **N**) the search in both directions. The **/**, **n** and **.** commands form a very useful trio for interactive replacement work.

The ex Mode is also used for substitution (**:s**). Both search and replace operations also use *regular expressions* for matching multiple patterns.

Test Your Understanding

7.1 You pressed **50k** to move the cursor 50 lines up but you see **50k** input as text. What mistake did you make and how do you remove the three characters?

7.2 How will you replace has with have in the current line?

7.3 How will you insert a line (i) above the current line, (ii) below the current line?

7.4 How do you abort an editing session?

7.5 Name three ways exiting a **vi** session after saving your work.

7.6 How will you quickly move to the fifth word of a line and replace its four characters with the string counter?

7.7 Find out the number of words in this string as interpreted by (i) **vi**, (ii) wc—29.02.2000 is_last_day_of_February.

7.8 In the current line, how do you take your cursor to (i) the 40th character, (ii) the beginning, (iii) the end?

7.9 Explain which of the following commands can be repeated or undone: (i) **40k** (ii) *[Ctrl-f]* (iii) **5x** iv)

7.10 You have wrongly entered the word Comptuer. How will you correct it to Computer?

7.11 From a conceptual point of view, how are **d** and **y** different from Command Mode commands like **J** and **$**?

7.12 How do you combine five lines into a single line?

7.13 How will you search for a pattern `printf` and then repeat the search in the opposite direction the original search was made?

7.14 Every time you press a **.** (dot), you see a blank line inserted below your current line. Why does that happen?

Flex Your Brain

7.1 Name the three modes of **vi** and explain how you can switch from one mode to another.

7.2 How will you add /* at the beginning of a line and */ at the end?

7.3 How do you remove the characters that you just inserted above without using the undo feature?

7.4 How do you move to line number 100 and then write the remaining lines (including that line) to a separate file?

7.5 **vi** refuses to quit with **:q**. What does that indicate and how do you exit anyway?

7.6 Explain what the following commands do: (i) **:.,10w foo** (ii) **:$w! foo**. In which mode are the commands executed and what difference does it make if foo exists?

7.7 Assuming that the cursor is at the beginning of the line, name the commands required to replace (i) `echo 'Filename: \c'` with `echo -n "Filename: "` (ii) `printf("File not found\n");` with `fprintf(stderr, "File not found\n");`

7.8 In the midst of your work, how can you see the list of users logged in? If you have a number of UNIX commands to execute, which course of action will you take?

7.9 Name the sequence of commands to execute to move to the line containing the string `#include`, deleting four lines there, and then placing the deleted lines at the beginning of the file.

7.10 Mention the sequence of commands to execute that will replace `printf(` with `fprintf(stderr,`? How will you repeat the action globally?

7.11 How do **u** and **U** differ? When will **U** fail to work?

7.12 Name the commands required to noninteractively replace all occurrences of cnt with count in (i) the first 10 lines, (ii) the current line, (iii) all lines. How do you repeat the exercise in an interactive manner?

7.13 If the power to the machine is cut off while a **vi** session is active, how does it affect your work? What salvage operation will you try?

7.14 You made some changes to a read-only file and then find that you can't save the buffer. What course of action will you take without quitting the editor?

7.15 Copy /etc/passwd to passwd. Name the **vi** commands required to save the first 10 lines in passwd1, the next 10 in passwd2 and the rest in passwd3.

7.16 List the editing and navigation commands required to convert the following text:

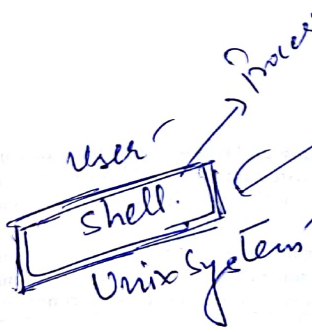
```
# include<errno.h>
void quit (char *message)
{
        printf("Error encountered\n");
        printf("error number %d, ", errno);
```

Scanned by CamScanner

```
        printf("quitting program\n");
        exit(1);
}
```

to this:

```
#include <stdio.h>
#include <errno.h>
void quit (char *message, int exit_status) {
        /* printf("Error encountered\n"); */
        fprintf(stderr, "Error number %d, quitting program\n", errno);
        exit(exit_status);
}
```



8

The Shel

This chapter introduces the agency that sits between the user and the UNIX system. It is calle *shell*. All the wonderful things that you can do with UNIX are possible because the shell doe of work on our behalf that could be tedious for us to do on our own. The shell looks for special symbols in the command line, performs the tasks associated with them and finally ex the command. For example, it opens a file to save command output whenever it sees the > s

The shell is a unique and multi-faceted program. It is a command interpreter and a progra language rolled into one. From another viewpoint, it is also a process that creates an envir for you to work in. All of these features deserve separate chapters for discussion, and you'll shell discussed at a number of places in this book. In this chapter, we focus on the she interpretive activities. We have seen some of these activities in previous chapters (like `ls | wc`), but it is here that we need to examine them closely.

What You Will Learn

- An overview of the shell's interpretive cycle.
- The significance of *metacharacters* and their use in *wild-cards* for matching multiple
- The use of *escaping* and *quoting* to remove the meaning of a metacharacter.
- The significance of the three *standard files (streams)* that are available to every con
- How the shell manipulates the default source and destination of these streams to *redirection* and *pipelines*.
- Understand what *filters* are and why they are so important in UNIX.
- The significance of the files `/dev/null` and `/dev/tty`.
- The use of *command substitution* to obtain the arguments of a command from t output of another.
- Shell variables and why they are so useful.

Scanned by CamScanner