

## Module 3

### The vi editor:

The vi editor was created Bill Joy for the BSD system. This program is now standard on all UNIX systems.

vi uses a number of internal commands to navigate to any point in a text file and edit the text there.

It allows us to copy and move text within a file and also from one file to another.

### vi Basics:

We can invoke vi with the filename.

### **Ex:**

```
vi sometext
```

The file may not exist earlier. The vi presents us 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 ~.

We will not be able to take the cursor there yet; because they are nonexistent lines.

The last line is reserved for commands that we enter to act on text. This line is also used by the system to display messages.

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

We cannot use the Command Mode to enter or to replace text.

To enter text, we must switch to Input Mode, by pressing the key marked i. Once we are in the Input Mode, whatever key we press will be displayed on the screen.

After text entry is complete, the cursor is positioned on the last character of the line. This is known as the current line and the character where the cursor is stationed is the **current cursor position**.

Now, if we press the [Esc] key, we will revert to Command Mode. If we press [Esc] key again and we'll hear a beep, which indicates that key has been pressed unnecessarily.

The file `sometext` does not exist yet. Actually, the text that we have entered has not been saved on disk but exists in some temporary storage called buffer.

To save the entered text, we must switch to the **ex Mode** (also known as the Last Line Mode).

We can invoke the ex Mode from the Command Mode by entering a `:` (colon), which shows up in the last line. We can enter an `x` and press [Enter.]

The file will then be saved on disk and `vi` returns the shell prompt. To modify this file, we will have to invoke **`vi sometext`** again.

The following are the three modes used by `vi`:

- **Command Mode**  
This is the default mode of the editor where every key pressed is interpreted as a command to run on text. We will have to be in this mode to copy and delete text.
- **Input Mode**  
Every key pressed after switching on this mode actually shows up as text.
- **ex Mode (Last Line Mode)** - This mode is used to handle files (like saving) and perform substitution. Pressing a `:` in the Command Mode invokes this mode. We can enter an ex Mode command followed by [Enter]. After the command is run, we are back to the default Command Mode.

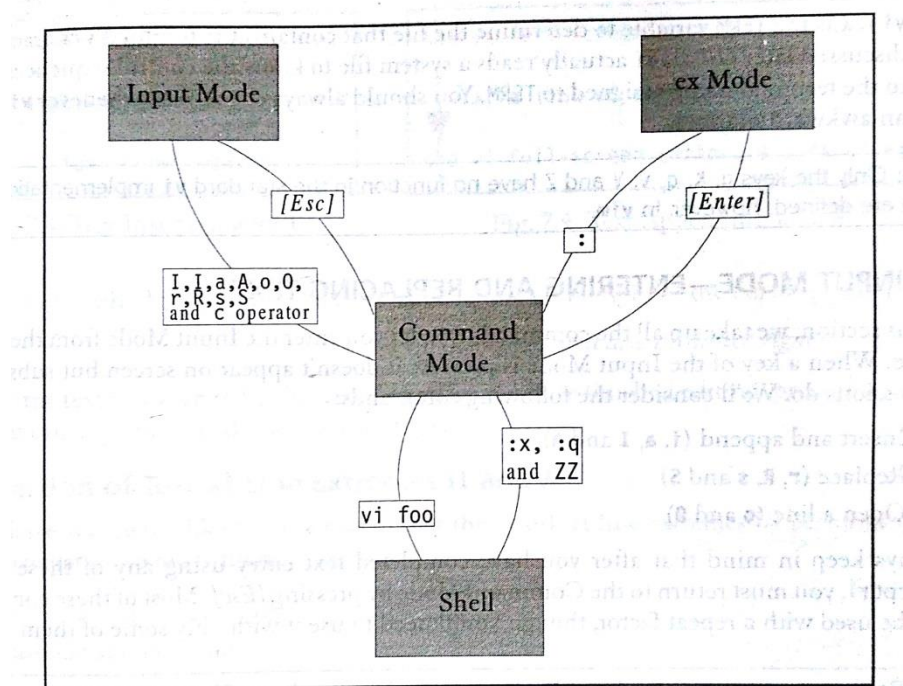


Fig. 7.2 The Three Modes

## **The file .exrc:**

The vi reads the file \$HOME/.exrc on startup.

Many ex Mode commands can be placed in this file so they are available in every session. We can create abbreviations, redefine our keys to behave differently and also make variable settings. The .exrc file will progressively develop into an exclusive “library” containing all shortcuts and settings that we use regularly.

## **INPUT MODE – ENTERING AND REPLACING TEXT:**

The following are Input Mode Commands:

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

After we have completed text entry using any of these commands (except r), we must return to the Command Mode by pressing [Esc].

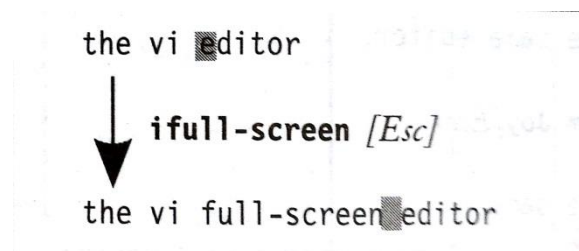
### **Insertion of Text (i and a):**

The simplest type of input is insertion of text. We just press **i**

Pressing this key changes the mode from Command to Input. The key depressions will result in text being entered and displayed on the screen.

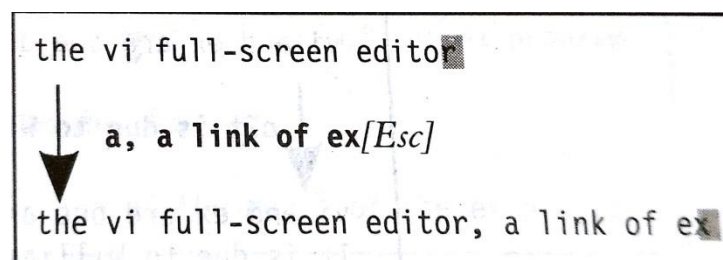
If the I command is invoked with cursor positioned on existing text, text on its right will be shifted further without being overwritten.

The insertion of text with **i** is as shown in the following figure, along with the position of the cursor.



There are other methods of inputting text.

To append the text to the right of the cursor position, we use **a** , followed by the text we wish to key in.



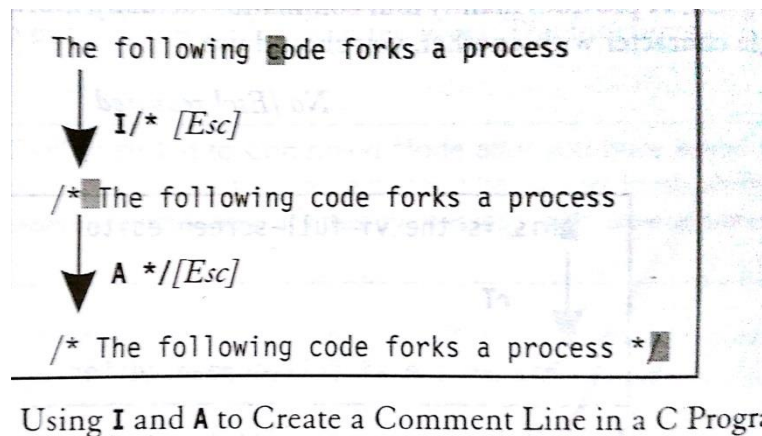
After we have finished editing, press [Esc].

### **Insertion of Text at Line Extremes (I and A):**

**I** - Inserts text at beginning of line.

**A** – Appends text at the end of line.

These two commands are suitable for converting code to comment lines in a C program as shown in the following figure:



A comment line in C is of the form `/*comment*/`.

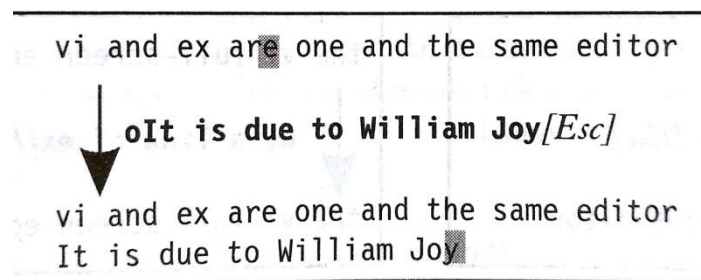
Use **I** on an existing line that you now wish to convert to a comment, and then enter the symbols `/*`.

After pressing [Esc], use **A** to append `*/` at the end of the line and press [Esc] again.

### **Opening a New Line (o and O):**

In **vi**, we use **o** and **O** to open a line.

**o** inserts an empty line below the current line as shown in the figure below:



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

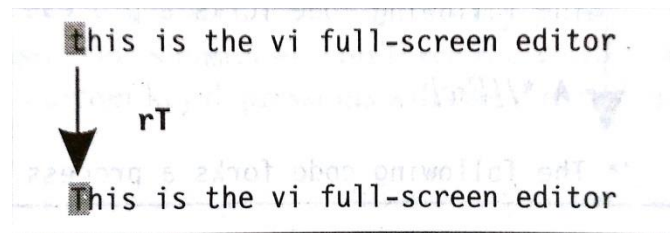
**O** also opens a line but above the current line.

### Replacing Text (r, s, R and S):

To replace a single character with another, we should use

**r** No [Esc] required

followed by the character that replaces the one under the cursor. Only a single character can be replaced 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.



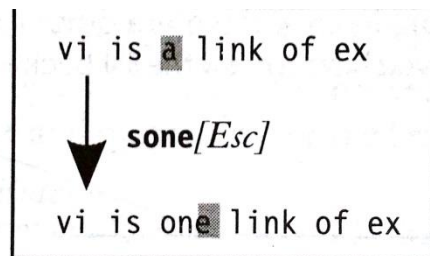
**Fig. 7.7** Replacing a Single Character with **r**

There is no need to press [Esc] when using **r** and the replacement character.

When we want to replace one character with many characters, we need to 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.

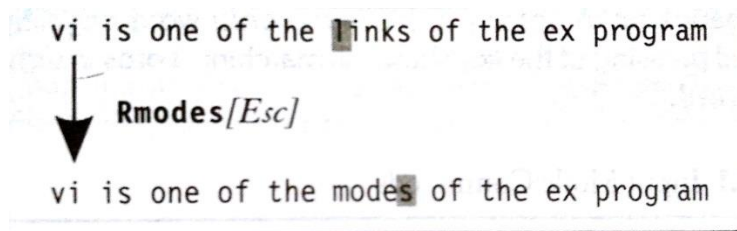


**Fig. 7.8** Replacing Text with **s**

**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)



**Fig. 7.9** Replacing Text with R

**Table 7.1** Input Mode Commands

<i>Command</i>	<i>Function</i>
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 <i>ch</i> (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

### **The ex Mode Commands:**

When we edit a file using vi, the original file is not disturbed , but only a copy of it is replaced in a **buffer**. From time to time, we should save our work by writing the buffer contents to disk to keep the disk file current. When we talk of saving a file, we actually mean saving this buffer. We also need to quit **vi** after or without saving the buffer. These features are handled by the ex Mode.

### **Saving Our Work (:w)**

We can save the buffer and remain in the editor. From time to time, we must use the **:w** command to write the buffer to disk.

We need to enter a **:**, which appears on the last line of the screen , then **w** and finally [Enter]:

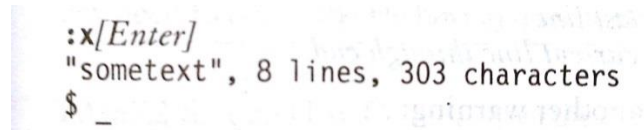
```
:w[Enter]
"sometext", 8 lines, 275 characters
```

We can now continue our editing work normally.

With the `:w` command we can optionally specify a filename as well. In that case, the contents are separately written to another file.

### **Saving and Quitting (:x and :wq)**

The previous command returns to the Command Mode so that we 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
$ _
```

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

### **Aborting Editing (:q)**

It is 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]           Won't work if buffer is unsaved
$ _
```

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

No write since last change (:quit! overrides)

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

```
:q!                  Ignores all changes made and quits
```

to return to the prompt irrespective of the status of the buffer.

### **Writing Selected Lines:**

The `:w` command is an abbreviated way of executing the ex Mode instruction `:l, $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`



We can save a single line as well:

**:5w n2words.pl** *Writes 5<sup>th</sup> line to another file*

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. We can use them singly or in combination:

<code>:.w tempfile</code>	<i>Saves current line (where cursor is positioned)</i>
<code>:\$w tempfile</code>	<i>Saves last line</i>
<code>:\$w tempfile</code>	<i>Saves current line through end</i>

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

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

The message is clear: **vi** is telling us to suffix the ! to the **:w** command to overwrite tempfile.

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

If we want to edit and compile our C program repeatedly, we 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**

\$\_

This returns a shell prompt. Execute cc or any UNIX command here and then return to the editor using [Ctrl-d] or exit.

### **Recovering from a Crash (:recover and -r)**

Accidents can and will happen, the power can go off, leaving work unsaved. However **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 lets this swap file remain on disk. **vi** will then complain the next time we invoke it with same file.

The complaint usually also contains some advice regarding the salvage operation.

We will be advised to use either the ex Mode command **:recover**, or **vi -r foo** to recover as much of foo as possible.



**Table 7.2** Save and Exit Commands of the ex Mode

<i>Command</i>	<i>Action</i>
:w	Saves file and remains in editing mode
:x	Saves file and quits editing mode
:wq	As above
:w n2w.pl	Like <i>Save As</i> ..... 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,n2w build.sql	Writes lines <i>n1</i> to <i>n2</i> 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 <i>cmd</i> command and returns to Command Mode
:sh	Escapes to UNIX shell
:recover	Recovers file from a crash

## **The Command Mode:**

This is the mode we come to when we have finished entering or changing our text.

A Command Mode command does not show up on screen but simply performs a function.

## **Navigation:**

### **Movements 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 each other in the middle row of the keyboard.

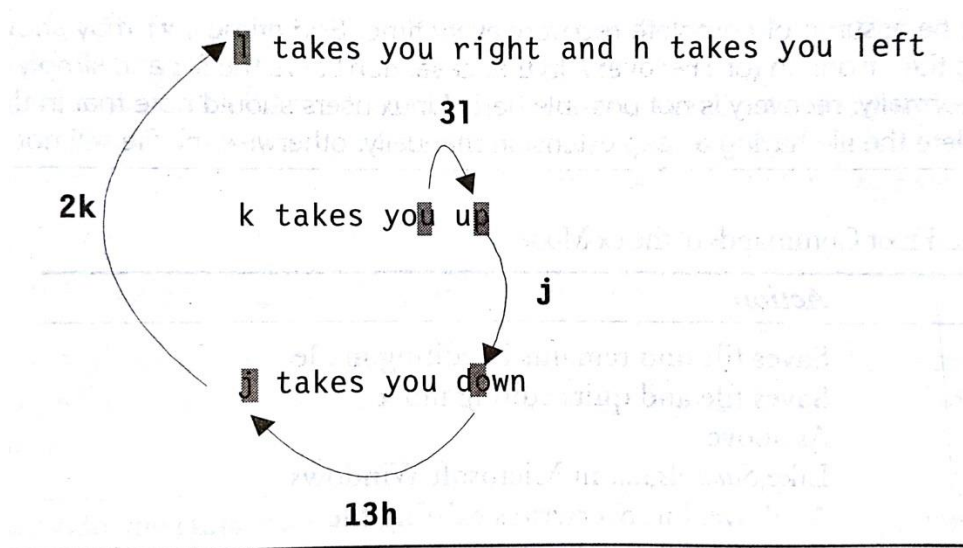
Without the repeat factor, they move the cursor by one position.

We can use the following keys for moving the cursor vertically:

- k Moves cursor up
- j Moves cursor down

To move the cursor along a line, we use the following commands:

- h Moves cursor left
- l Moves cursor right



**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 line sup and **20h** takes it 20 characters to the left.

We can note that this motion is relative, i.e, we cannot move to a specific line number with these keys.

### **Word Navigation (*b*, *e* and *w*):**

**vi** understands a word as a navigation unit which can be defined in two ways, depending on the key presses.

If our cursor is a number of words away from our desired position, we 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 the 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 three words forward.

A word here is simply a string of alphanumeric characters and the **\_** (underscore).

Ex: **Bash** is one word

**sh\_profile** is one word

**tcp-ip** is three words

### **Moving to the Line Extremes (0, | and \$ ):**

**0** (zero) or **|** is used to move to the first character of a line.

The **|** takes a repeat factor and using that, we can position the cursor on a certain column.

Ex: **30|** moves cursor to column 30

The **\$** is used to move to the end of the current line.

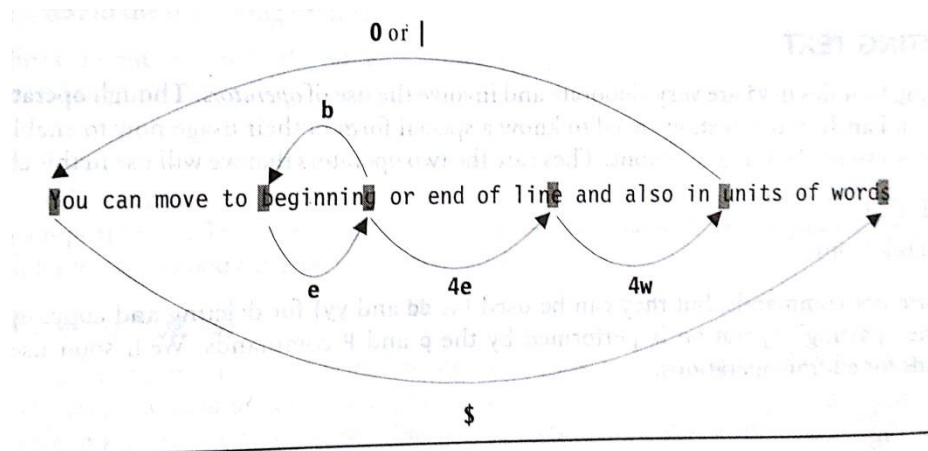


Fig. 7.11 Navigation Along a Line

### **Scrolling ([Ctrl-f], [Ctrl-b], [Ctrl-d] and [Ctrl-u]) :**

The two commands for scrolling a page at a time are

[Ctrl-f]        Scrolls forward

[Ctrl-b]        Scrolls backward

We can use the repeat factor , like in **10[Ctrl-f]**, to scroll 10 pages and navigate faster in the process.

[Ctrl-u]      Scrolls half page backward

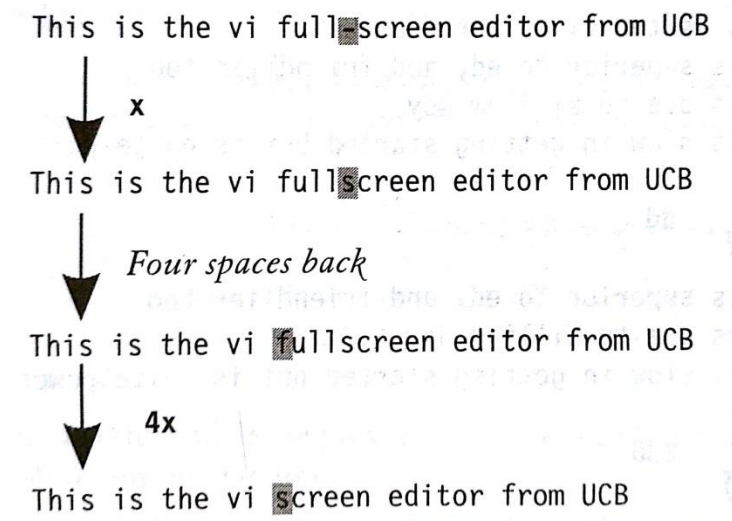
At any time, we can press [Ctrl-g] to know the current line number:

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

<b>x</b>	<i>Deletes a single character</i>
----------	-----------------------------------

The character under the cursor gets deleted, and the text on the right shifts left to fill up the space.

A repeat factor can also be used, so **4x** deletes the current character as well as three characters from the right.



**Fig. 7.12** Deleting Text with **x**

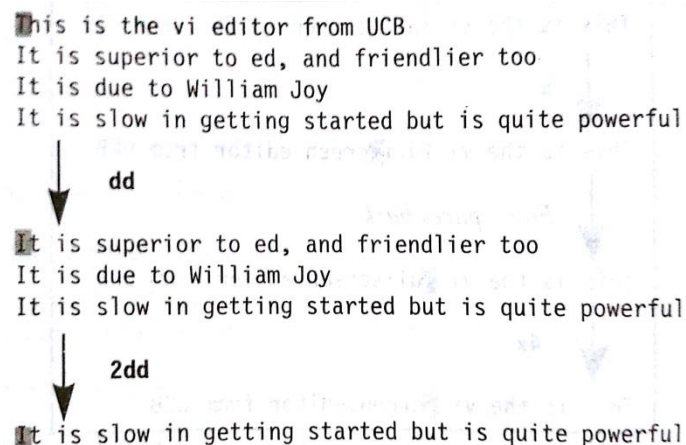
Deletion of text from the left is handled by the **X** command.

If we keep it pressed, we will see that we have erased all text to the beginning of the line.

Entire lines are removed with the **dd** command. We can 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.

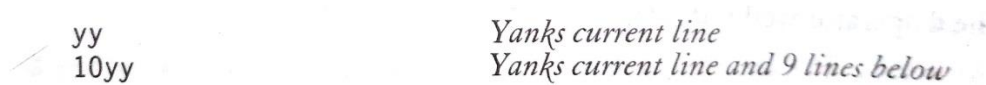


**Fig. 7.13** Deleting Lines with **dd**

## **Copying Text (y and p)**

**vi** uses the term yanking for copying text, the reason why the operator is named **y**.

To copy (or yank) one or more lines, we use the **yy** command:



This yanked text has to be placed at the new location.

The put commands, **p** and **P** can be used to paste the copied lines.

## **Joining Lines (J)**

To join the current line and the line following it, we use

**J**

**J** removes the newline character between the two lines to pull up the line below it.

## **Undoing Last Editing Instructions (u and U):**

We can revert the last change we have made to the buffer by pressing

**u**

This will undo the most recent single editing change by restoring the position before the change.

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

The command

**U**

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

## **Repeating the Last Command . (dot)**

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.

### Example:

- 1) If we have deleted two lines of text with **2dd**, then to repeat this operation elsewhere, all we have to do is to position the cursor at the desired location and press

.

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

- 2) Consider that we have to indent a group of lines by inserting tab at the beginning of each line.

We need to use *i[Tab][Esc]* only once, say on the first line. We 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 . command can be used to repeat only the most recent editing operation – insertion, deletion or any other action that modifies the buffer.

### Searching for a Pattern (/ and ?)

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 we are looking for the string **printf**, we can enter this string after the /:

**/printf [Enter]**

The search begins forward to position the cursor on the first instance of the word.

**vi** searches the entire file, so if the pattern cannot 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

The sequence

**?pattern [Enter]**

searches backward for the most previous instance of the pattern.



## Repeating the last pattern search (n and N)

For repeating a search in the direction the previous search was made with / and ?, we use

**n**

to repeat the search in the same direction of the original search.

The cursor will be positioned at the beginning of the pattern. In this way, we can press **n** repeatedly to scan all instances of the string.

**N** reverses the direction pursued by **n**.

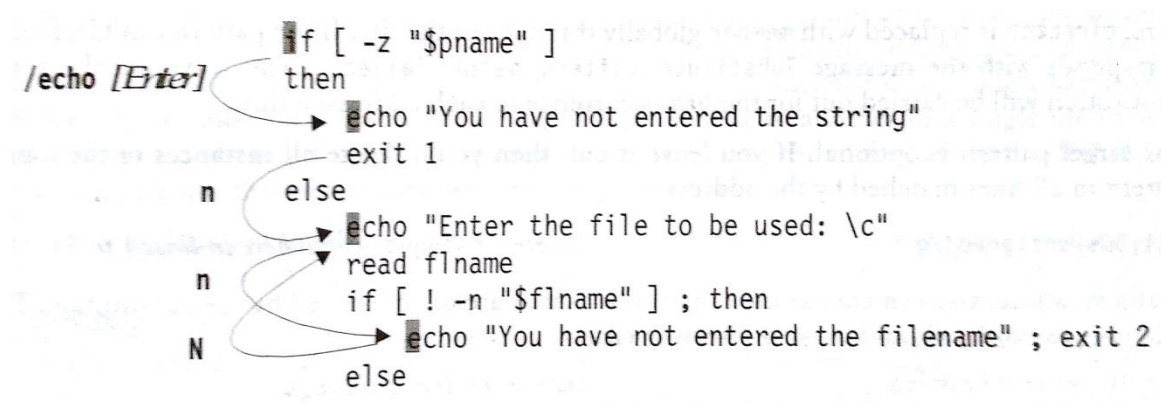


Fig. 7.14 Search and Repeat with / and n

Table 7.3 Search and Repeat Commands

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

## Substitution – Search and Replace (:s)

The substitution feature of **vi** is achieved in the ex-Mode using the **s** (substitute) command.

It lets us replace a pattern in the file with something else.

The syntax is as shown below:

**:address/source\_pattern/target\_pattern/flags**

The *source\_pattern* here is replaced with *target\_pattern* in all lines specified by *address*.

The *address* can be one or a pair of numbers, separated by a comma.

Ex:

***:1,\$s/director/member/g***

In the above sequence, 1,\$ addresses all lines in a file. The most commonly used flag is g, which carries out the substitution for all occurrences of the pattern in a line.

Here, *director* is replaced with *member* globally throughout the file. If the pattern cannot be found **vi** responds with the message

**Substitution pattern match failed.**

If we leave out the g, the substitution will be carried out for the first occurrence in each addressed line.

The target pattern is optional.

If we leave it out, then we will delete all instances of the source pattern in all lines matched by the address:

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

The above sequence deletes unsigned everywhere in lines 1 to 50

We can choose a range of lines that are affected by the substitution. Following are some examples:

<b><i>:3,10s/director/member/g</i></b>	<i>Substitute lines 3 through 10</i>
<b><i>:.s/director/member/g</i></b>	<i>Only the current line</i>
<b><i>:\$s/director/member/g</i></b>	<i>Only the last line</i>

*Interactive Submission:*

Sometimes we may like to selectively replace a string .

In that case, we need to add 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 our response:

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

A y performs the substitution, any other responses does not.

This sequence is repeated for each of the matched lines in turn.

### **Customizing vi :**

There are three ex Mode commands which lets us customize the vi editor:

1. set
2. map
3. abbr

vi reads the file ~/.exrc only once, upon startup , and its behavior is determined by statements placed in that file.

### **The set command:**

The general vi environment is determined by its variable settings.

These variables are controlled by the **:set** command.

**Ex:**

```
:set showmode
:set autowrite
```

**set** is used with a variable name(like showmode).

The variable in the second example takes on an abbreviation, which means that we can also use **:set aw**

Many of these variables can have the string **no** prefixed to their name, in which case the setting is deactivated.

For instance, **noautowrite** negates autowrite.

### **Automatic indentation (autoindent) :**

Programmers need to provide indentation to their code for easier readability.

This is taken care of when we use the following set statement:

**:set autoindent**

When this option is set, an [Enter] in the input Mode places the cursor in the next line at the current indentation.

### **Numbering Lines (number):**

To remove the line numbers from being displayed, we can use the following set command:

**:set nonumber**

### **Ignoring Case in Pattern Searches (ignorecase)**

The search commands may or may not pursue a case-insensitive search. It depends on the ignorecase setting.

By default, this is generally off, but we can turn it on with

**:set ignorecase**

### **No Regular Expressions in Search (nomagic)**

We can remove the meanings of the regular expression by using the following set command:

**:set nomagic**

## **Map: Mapping Keys of Keyboard:**

The map command lets us assign the undefined keys or reassign the defined ones so that when such a key is pressed, it expands to a command sequence.

The command **:map** is followed by the key which needs mapping and the key sequence which is mapped.

**Ex:**

**:map g :w^M**                      ^M signifies the [Enter] key

The above command condenses the **:w[Enter]**, to the character g.

## **Abbr: Abbreviating Text Input**

The **abbreviate** command is used to expand short strings to long words.

**Ex:**

**:ab re regular expression**

**:ab me metacharacter**

This indicates that when we enter the word re followed by a key which is neither alphanumeric nor the \_ (underscore) character, we will see re expanded to regular expression.

## The Shells interpretive cycle:

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and **waits** for you to enter a command.
- After the command is entered, the shell scans the command line for **metacharacters** and expands abbreviations (like the \* in rm \*) to recreate a simplified command line.
- It then passes on the command line to the **kernel** for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. Now the user is free to enter another command.

## Wild cards and file name generation:

- The metacharacters that are used to construct the generalized pattern for matching filenames are called **wild cards**.
- **Wild cards** are the set of characters that the shell uses to match **filenames**.
- The shell will expand all the **wild cards** before passing the command to the kernel for execution.
- The significance of the various metacharacters in the wild card set are listed in the table below:

Wild cards	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of the characters x and z
[!ijk]	A single character that is not i, j or k
[!x-z]	A single character that is not within the ASCII range of the characters x and z
{pat1, pat2}	pat1, pat2, etc

## EXAMPLES:

### ➤ The \* and ?

- The command **ls chap \*** to list some filenames beginning with **chap**.

```
$ ls chap*
```

```
chap  chap01  chap02  chap03  chap04  chap15  chap16  chap17
chapx  chapy  chapz
```

- The command **ls chap?** Matches five character filenames beginning with chap

```
$ ls chap?
```

```
chapx  chapy  chapz
```

( note: \* and ? doesn't match all files beginning with a .(dot) or the / of a pathname )

### ➤ **Matching the Dot:**

- The \* does not match all files beginning with a . (dot) or the / of a pathname.
- If we want to list all hidden filenames in our directory having atleast three characters after the dot, the dot must be matched explicitly.

```
$ ls .???
```

```
.bash_profile  .exerc  .netscape  .profile
```

### ➤ **The character class [ ] :**

- The character class allows the user to frame more restrictive patterns.
- The character class comprises a set of characters enclosed by the rectangular brackets, [ and ], but it matches a **single** character in the class
- The pattern [abcd] is a character class, and it matches a single character – an a, b, c, or d.

```
$ ls chap0[124]
```

```
chap01  chap02  chap04
```

- **Range specification** is also possible inside the class with a – (hyphen); the two characters on either side of it form the range of characters to be matched.

```
$ ls chap0[1-4]
```

```
chap01  chap02  chap04
```

```
$ls chap[x-z]
```

```
chapx  chapy  chapz
```



- **Negating the character class (!)** you can use the ! as the first character in the class to negate the class.

**\*.[!co]** Matches all filenames with a single character extension but not the .c or .o files

**[!a-zA-Z]\*** Matches all the filenames that don't begin with an alphabetic character.

- **Matching totally dissimilar patterns { }** : the dissimilar patterns should be written within the flower brackets { and } separated by comma.

**{c,java}** Matches the pattern either c or java

**{include,bin,lib}** Matches pattern either include, bin or lib.

### Examples:

<code>ls *.c</code>	Lists all files with extension .c.
<code>mv * ../bin</code>	Moves all files to bin subdirectory of parent directory.
<code>cp foo foo*</code>	Copies foo to foo* (* loses meaning here).
<code>cp ?????? progs</code>	Copies to progs directory all files with six-character names.
<code>lp note[0-1][0-9]</code>	Prints files note00, note01 ..... through note19.
<code>rm *,[11][10][19]</code>	Removes all files with three-character extensions except the ones with the .log extension.
<code>cp -r /home/kumar/{include,lib,bin} .</code>	Copies recursively the three directories, include, lib and bin from /home/kumar to the current directory. (Not in Bourne shell)

## Removing the special meaning of the wild cards: (Escaping and Quoting):

### ➤ Escaping

- Providing a \ (backslash) before the wildcard to remove (escape) its special meaning.
- For instance, in the pattern \\*, the \ tells the shell that the asterisk has to be matched literally instead of being interpreted as a metacharacter.

**rm chap\\***

*Doesn't remove chap1, chap2*

- The \ suppresses the wild-card nature of the \*, thus preventing the shell from performing the filename expansion on it.

**\$ ls chap0\[1-3\]**

Must escape the [ and ]

chap0[1-3]

- *Escaping the space*

**\$ rm My\ Document.doc**

without \ rm would see two files

- *Escaping the \ itself* sometimes we may need to interpret the \ itself literally. You need another \ before it, that's all:

**\$ echo \\**

\

**\$ echo the newline character is \n**

the newline character is \n

### ➤ Quoting

- Enclosing the wild-card, or even the entire pattern, within quotes, anything within these quotes are left alone by the shell and not interpreted.
- The following example shows the protection of the four special characters using single quotes:

**\$ echo 'the characters |, <, > and \$ are also special'**

the characters |, <, > and \$ are also special

- **Single quotes** protect all special characters ( except single quotes ).
- **Double quotes** are more permissive; they don't protect ( apart from double quotes itself) the \$ and ` (backquote)

## Redirection: The Three Standard Files

- The Shell associates **three files** with the terminal - two for the **display** and one for the **keyboard**.
- The command performs all the terminal – related activities with these three files that are provided by the shell.
- These special files are actually streams of characters which many commands see as input and output. A stream is simply a sequence of bytes.

### 1. Standard Input:

- The file (or stream) representing input, which is connected to the keyboard.
- When the commands are used without the filename arguments they read the file representing the **standard input**.

This file is indeed special; it can represent three input sources:

1. The **keyboard**, the default source.
2. A file using **redirection** with the < symbol ( a metacharacter ).
3. Another program using a **pipeline**.

When we use the **wc** command without an argument and have no special symbols < and | in the command line, **wc** obtains its input from the default source.

We have to provide this input from the keyboard and mark the end of input with [**Ctrl-d**].

```
$ wc
Standard input can be redirected
It can come from a file
or a pipeline
[Ctrl-d]
3      14      71
```

- **Redirection ( < )** : shell can reassign the standard input file to a disk file. This means it can redirect the standard input to originate from a file on disk. This reassignment or redirection requires the < symbol:

```
$ wc < sample.txt
3      14      71
```

- **wc** command didn't open sample.txt. It read the standard input file as a stream but only after the shell reassigned this stream to a disk file (sample.txt).

```
$ bc < math.txt
```

**bc** command will read input from the standard input which is assigned to math.txt using < symbol.

## 2. Standard output:

- The file (or stream) representing output, which is connected to the display.
- All commands displaying output on the terminal actually write to the standard output file as streams of characters.
- There are three possible destinations of this stream:
  1. The **terminal**, the default source.
  2. A file using **redirection** with the symbols > and >> ( a metacharacter ).
  3. As input to another program using a **pipeline**.
- **Redirection ( > and >> )** : shell can reassign the standard output to a disk file. This means the default destination can be replaced with any file by using > or >> operator, followed by the filename:

```
$ wc sample.txt > newfile
```

```
$ cat newfile
```

```
3      14      71 sample.txt
```

- The first command sends the word count of sample.txt to newfile; nothing appears on the terminal screen. If the output file doesn't exist, the shell creates it before executing the command.
- If it exists, the shell overwrites it.
- The shell also provides >> symbol to append to a file ( prevents overwriting ).

```
$ wc sample.txt >> newfile      Doesn't disturb existing contents
```

## 3. Standard error:

- Each of the three standard files is represented by a number, called a **file descriptor**.
- A file is opened by referring to its pathname, but subsequent read and write operations identify the file by this file descriptor.
- The first three slots are generally allocated to the three standard streams in this manner:

```
0 – Standard input
```

```
1- Standard output
```

```
2- Standard error
```

- These descriptors are implicitly prefixed to the redirection symbols. For instance > and 1> mean the same thing to the shell, while < and 0< are identical.
- We need to explicitly use one of these descriptors for handing the standard error stream.
- Redirecting the standard error requires the use of the 2> symbols:

```
$ cat foo 2> errorfile
```

```
$ cat foo
```

```
cat: cannot open foo
```

- You can append standard error to a file :

**\$ cat foo 2>> errorfile**

## Connecting commands: Pipe ( | ) :

- **Pipe** allows the standard input stream to connect with the standard output stream such that one command can take input from another.

- For example

**\$ who | wc -l** *counts the number of online users*  
5

- Here the shell connects the **who**'s standard output to **wc**'s standard input using a special operator called pipe ( | ).
- The output of **who** has been passed directly to the input of **wc**, and who is said to be **pipelined** to wc.
- When multiple commands are connected this way, a **pipeline** is said to be formed.
- It's the shell that set up the connection and the commands have no knowledge of it.
- One can count the number of files in a directory by combining **ls** and **wc -w** commands using a pipe.

**\$ ls | wc -w** *counts the number of files and subdirectories*  
15

- There is no restriction in the number of commands you can use in a pipeline.

## Splitting the output: tee

- **tee** is an external command, it handles a character stream by duplicating its input.
- It saves one copy in a file and writes the other to standard output.
- **tee** can be placed anywhere in a pipeline.
- The following command sequence uses tee to display the output of who and saves this output in a file as well:

```
$ who | tee user.txt
Romeo      pts/2      sep 7 08:41
Juliet     pts/3      sep 7 17:58
Sumit      pts/5      sep 7 18:01
```

- **user.txt** also contains this output.
- Consider the following example

**\$ who | tee user.txt | wc -l**  
3

- One copy of the output of who is saved in **user.txt** and another copy is given to standard output through **pipe** to the **wc -l** command which displays the total number of users online.

## Command Substitution:

- Shell enables one or more command **arguments** to be obtained from the standard output of another command. This feature is called as **command substitution**.
- The command substitution is done using the special character ` (backquote), the command that need to be substituted should be written within the backquotes.
- The date command's output is incorporated into the echo statement using the command substitution as follows

```
$ echo the date today is `date`
```

```
the date today is Sat sep 7 19:01:16 IST 2016
```

- The shell executes the command enclosed in backquotes first and replaces the enclosed command line with the output of the command.

```
$ echo "there are `ls | wc -w` files in the current directory"
```

```
there are 50 files in the current directory
```

- The `ls | wc -w` command's output is placed in the echo commands argument.

Consider a file emp.lst that contains the database of employees as shown below:

```
$ cat emp.lst
2233|a.k. shukla      |g.m.    |sales    |12/12/52|6000
9876|jai sharma        |director|production|12/03/50|7000
5678|sumit chakrobarty|d.g.m.  |marketing|19/04/43|6000
2365|barun sengupta    |director|personnel |11/05/47|7800
5423|n.k. gupta        |chairman|admin     |30/08/56|5400
1006|chanchal singhvi  |director|sales     |03/09/38|6700
6213|karuna ganguly    |g.m.    |accounts |05/06/62|6300
1265|s.n. dasgupta     |manager |sales     |12/09/63|5600
4290|jayant Choudhury  |executive|production|07/09/50|6000
2476|anil aggarwal     |manager |sales     |01/05/59|5000
6521|lalit chowdury    |director|marketing |26/09/45|8200
3212|shyam saksena     |d.g.m.  |accounts |12/12/55|6000
3564|sudhir Agarwal    |executive|personnel |06/07/47|7500
2345|j.b. saxena       |g.m.    |marketing |12/03/45|8000
0110|v.k. agrawal      |g.m.    |marketing |31/12/40|9000
```

## grep: Searching for a Pattern:

The **grep** command scans its input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs.

### Syntax:

**grep options pattern filename(s)**

**grep** searches for pattern in one or more filename(s), or the standard input if no filename is specified.

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

Since **grep** is also a filter, it can search its standard input for the pattern, and also save the standard output in a file:

**who | grep kumar > foo**

**grep** also silently returns the prompt in case the pattern cannot be located.

When **grep** is used with multiple filenames, it displays the filenames along with the output.

### Ex:

```
$ grep "director" emp1.lst emp2.lst
emp1.lst:1006|chanchal singhvi |director |sales    |03/09/38|6700
emp1.lst:6521|lalit chowdury    |director |marketing|26/09/45|8200
emp2.lst:9876|jai sharma      |director |production|12/03/50|7000
emp2.lst:2365|barun sengupta    |director |personnel |11/05/47|7800
```

Quoting is essential when the pattern contains multiple words:

```
$ grep 'jai sharma' emp.lst
9876|jai sharma      |director |production|12/03/50|7000
```



## grep Options:

**Table 13.1** Options Used by **grep**

<i>Option</i>	<i>Significance</i>
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e <i>exp</i>	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen.
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f <i>file</i>	Takes patterns from <i>file</i> , one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in <b>fgrep</b> -style)

### Ignoring Case (-i):

When we look for a name but are not sure of the case, we can use the -i (ignore) option.

This option ignores case for pattern matching:

**Ex:**

```
$ grep -i 'agarwal' emp.lst
3564|sudhir Agarwal |executive|personnel |06/07/47|7500
```

### Deleting Lines (-v):

The -v option selects all the lines except those containing the pattern.

This option is frequently used with redirection.

**Ex:**

We can create a file otherlist containing all but directors:

```
$ grep -v 'director' emp.lst > otherlist
$ wc -l otherlist
11 otherlist
```

*There were 4 directors initially*

### **Displaying Line numbers (-n):**

The -n (number) option displays the line numbers containing the pattern, along with the lines:

```
$ grep -n 'marketing' emp.lst
3:5678|sumit chakrobarty|d.g.m. |marketing |19/04/43|6000
11:6521|lalit chowdury |director |marketing |26/09/45|8200
14:2345|j.b. saxena |g.m. |marketing |12/03/45|8000
15:0110|v.k. agrawal |g.m. |marketing |31/12/40|9000
```

The line numbers are shown at the beginning of each line, separated from the actual line by a :

### **Counting Lines Containing Pattern (-c):**

The -c (count) option counts the number of lines containing the pattern (which is not the same as number of occurrences).

**Ex:**

```
$ grep -c 'director' emp.lst
4
```

If we use this option with multiple files, the filename is prefixed to the line count:

```
$ grep -c director emp*.lst
emp.lst:4
emp1.lst:2
emp2.lst:2
empold.lst:4
```

### **Displaying Filenames (-l):**

The -l (list) option displays only the names of files containing the pattern.

**Ex:**

```
$ grep -l 'manager' *.lst
desig.lst
emp.lst
emp.lst
emp.lst
```

### Matching Multiple Patterns (-e):

The `-e` option can be used to match multiple patterns.

Ex:

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

### Taking patterns from a file (-f):

We can place all the patterns in a separate file, one pattern per line.

grep uses the `-f` option to take pattern from a file:

Ex:

```
grep -f pattern.lst emp.lst
```

### Basic Regular Expressions (BRE):

Table 13.2 The Basic Regular Expression (BRE) Character Subset

<i>Symbols or Expression</i>	<i>Matches</i>
<code>*</code>	Zero or more occurrences of the previous character
<code>g*</code>	Nothing or g, gg, ggg, etc.
<code>.</code>	A single character
<code>.*</code>	Nothing or any number of characters
<code>[pqr]</code>	A single character <i>p</i> , <i>q</i> or <i>r</i>
<code>[c1-c2]</code>	A single character within the ASCII range represented by <i>c1</i> and <i>c2</i>
<code>[1-3]</code>	A digit between 1 and 3
<code>[^pqr]</code>	A single character which is not a <i>p</i> , <i>q</i> or <i>r</i>
<code>[^a-zA-Z]</code>	A nonalphabetic character
<code>^pat</code>	Pattern <i>pat</i> at beginning of line
<code>pat\$</code>	Pattern <i>pat</i> at end of line
<code>bash\$</code>	bash at end of line
<code>^bash\$</code>	bash as the only word in line
<code>^\$</code>	Lines containing nothing

The regular expressions are a feature of the command that uses it and has nothing to do with the shell.

Regular expressions take care of some common query and substitution requirements.

POSIX identifies regular expressions as belonging to two categories:

1. Basic
2. Extended.

The grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the -E option.

### **The Character Class:**

A regular expression lets us specify a group of characters enclosed within a pair of rectangular brackets, [ ], in which case the match is performed for a single character in the group.

Thus the expression

**[ra]**

Matches either an r or an a.

The metacharacters [ and ] can now be used to match Agarwal and agrawal.

The following regular expression

**[aA]g[ar][ar]wal**

matches the two names.

The character class [aA] matches the letter a in both lowercase and uppercase.

The model [ar][ar] matches any of the four patterns:

aa      ar      ra      rr

Ex:

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

### **Negating a Class (^):**

Regular expressions uses the ^ (caret) to negate the character class, while the shell uses the ! (bang).

When the character class begins with a caret, all the characters other than the ones grouped in the class are matched.

So [^a-zA-Z] matches a single non-alphabetic character string.

### **The \*:**

The \* (asterisk) refers to the immediately preceding character.

It indicates that the previous character can occur many times, or not at all.

The pattern

`g*`

matches the single character `g`, or any number of `gs`.

Because the previous character may not occur at all, it also matches a null string.

It also matches the following stringd:

`g      gg      ggg      gggg.....`

It marks the zero or more occurrences of the previous character.

To match a string beginning with `g`, we should use `gg*` instead of `g*`.

### **Ex:**

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

The above regular expression can be used instead of using the `-e` option of **grep**.

### **The Dot:**

The . (dot matches a single character).

The pattern

`2...`

Matches a four character pattern beginning with a `2`.

### ***The Regular Expression .\****

It signifies any number of characters or none.

Ex: Consider that we want to look up the name **j. saxena** but are not sure whether it actually exists in the file as **j.b. saxena** or as **joginder saxena**.

```
$ grep "j.*saxena" emp.lst
2345|j.b. saxena      |g.m.      |marketing |03/12/45|8000
```

### Specifying the Pattern Locations (^ and \$):

Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurrence only at a particular location.

The following are the two characters that are used:

^ (caret) – For matching at the beginning of a line

\$ - For matching at the end of a line.

Ex:

In the above mentioned file, if we want to extract those lines where the emp-id begins with a 2, the

2.. shall not work.

We must indicate to grep that the pattern occurs at the beginning of the line and the ^ does it easily:

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

Similarly, to select those lines where the salary lies between 7000 and 7999, we have to use the \$ at the end of the pattern:

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

Similarly for listing only the directories we can use:

```
ls -l | grep "^d"
```

### **Escaping the Metacharacters:**

We can use the \ for escaping.

Ex: When are looking for a pattern `g*`, we can use `grep "g\*"`

### **Extended Regular Expression:**

The Extended Regular Expression (ERE) make it possible to match dissimilar patterns with a single expression.

EREs when compared with BREs form very powerful regular expressions.

#### ***The + and ?***

The ERE set includes two special characters , + and ?.

They signify the following:

+ Matches one or more occurrences of the previous character

? Matches zero or one occurrences of the previous character

- In both cases, the emphasis is on the previous character.
- This means that `b+` matches `b`, `bb`, `bbb`, etc., but unlike `b*`, it does not match nothing.
- The expression `b?` matches either a single instance of `b` or nothing.

Ex:

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

In the above example, `gg?` restricts the expansion to one or two `gs` only.

The + character can be used when we are looking for a multiword string like `#include<stdio.h>`, but do not know how many spaces separate the `#include` and `<stdio.h>`.

We can use the expression `#include +<stdio.h>` to match them all.

This expression matches the following patterns:

```
#include <stdio.h>  #include      <stdio.h>  #include          <stdio.h>
```



## Matching Multiple Patterns (|, ( and )):

The `|` is the delimiter of multiple patterns.

Using it, we can locate both sengupta and dasgupta from the file and without using the `-e` option twice:

```
$ grep -E 'sengupta|dasgupta' emp.lst
2365|barun sengupta      |director |personnel |11/05/47|7800
1265|s.n. dasgupta      |manager  |sales     |12/09/63|5600
```

The ERE thus handles the problem easily.

The characters `(` and `)`, lets us group patterns, and when you use the `|` inside the parantheses, we can frame an even more compact pattern:

```
$ grep -E '(sen|das)gupta' emp.lst
2365|barun sengupta      |director |personnel |11/05/47|7800
1265|s.n. dasgupta      |manager  |sales     |12/09/63|5600
```

Summary of ERE:

<i>Expression</i>	<i>Significance</i>
<i>ch+</i>	Matches one or more occurrences of character <i>ch</i>
<i>ch?</i>	Matches zero or one occurrence of character <i>ch</i>
<i>exp1 exp2</i>	Matches <i>exp1</i> or <i>exp2</i>
<i>GIF JPEG</i>	Matches <i>GIF</i> or <i>JPEG</i>
<i>(x1 x2)x3</i>	Matches <i>x1x3</i> or <i>x2x3</i>
<i>(lock ver)wood</i>	Matches <i>lockwood</i> or <i>verwood</i>