CHAPTER W25

Editing Text Files

**Objectives**

\* To explain the general utility of editing text files on a Linux system

\* To show the basic capabilities of vi, vim, and gvim

\* To show the basic capabilities of GNU emacs

\* To illustrate some of the important ways of customizing these editors

\* To cover the commands and primitives  
cp, emacs, gvim, ls, pwd, sh, vi, vim, who

W25.1 INTRODUCTION AND QUICK START

Question: What is the salient difference, for a novice Linux user, between a “text editor” and a “word processor”?

Answer: Simply put, text editing is Character User Interface (CUI)-oriented, and word processing is Graphical User Interface (GUI)- oriented.

Following up on, and greatly expanding the coverage provided in Chapter 3 of the printed book on Linux text editors, we expose the ordinary novice Linux user to basically two other families of text editor in this chapter: vi and emacs. They are historically grounded in UNIX, when that operating system was purely text-based (similar to its extremely developed command line interface), and with the advent of more sophisticated word processors in GUI environments, UNIX (and then Linux) text editors took on equivalent trappings in the mid-1980’s. But Linux text editors never dispensed with a purely text-based mode of operation, as can be seen in the sections of this chapter. We use the following editors that are commonly available in our reference Debain-family and CentOS modern Linux systems:vi, vim, gvim, and GNU emacs.

W25.1.1 Quick Start: The Simplest Path through These Editors

To stress how the keyboard keys are used in these editors, we provide the following reference to the keys used to execute commands or change modes:

1. Pressing the Escape key is signified as <Esc>

2. Pressing the Enter key is signified as <Enter>

W25. Pressing the <Ctrl> key in combination with another single key is signified as <Ctrl+X>, where you hold down the <Ctrl> key and press the X key (or any valid key for that combination) at the same time.

4. Pressing the Alt key in combination with another single key is signified as <Alt+X>,

where you hold down the <Alt> key and press the x key (or any valid key for that combination) at the same time.

5. A variant of 3. and 4. is shown as <Ctrl+X> a [b], where you first press and release <Ctrl> and x simultaneously, then press the a key, and optionally press the b key (or any valid combination of single keys or strings of characters).

6. In GNU emacs for Linux , the Meta key that is referred to in much of the literature on GNU emacs is the <Alt> key.

What you type, or hold down on the keyboard, is shown in **bold** text.

For vi, vim, and gvim

* At the shell prompt, run the program by typing **vi file1** then press **<Enter>**.
* Type **A**.
* Type some text.
* Press **<Esc>**.
* Type **:** (colon).
* Type **wq** then press **<Enter>**.

You now have a file in your default directory named **file1** with the text you typed in it.

If GNU emacs is not installed on your system, skip ahead to the next subsection for general instructions on how to install it. Then do the following:

For GNU emacs

* At the shell prompt, run the program by typing **emacs file2** then press **<Enter>**.
* Type some text.
* Hold down **<Ctrl+U>**, then **<Ctrl+X>**, then **<Ctrl+C>**.

You now have a file in your default directory named **file2** with the text you typed in it.

W25.1.2 First Comments on Linux Editors

As you can see from Section W25.1.1, with vi, vim, and gvim, you can’t immediately begin to enter text into the file you are editing. You have to be in *Insert mode* to do that; that’s what typing A as the second step is doing. Vi, vim, and gvim have modes.

In GNU emacs, you can start typing text into the file immediately. Emacs is a *modeless* editor. That basically means that you can immediately begin entering and editing text using the keyboard and some particular pointing device, such as a mouse or keyboard arrow keys.

We present the tutorial information in this chapter using typed commands, and by using graphical modes of input and editing.

It is very important to realize that vi, vim, and gvim all generally use the same commands and have basically the same functionality. But vim and gvim are not only more graphical - allowing you to work more efficiently in GUI environments such as those on our base modern Linux systems - but they also have an improved and expanded command structure. This will become more evident to you, for example, in Section W25.2.9, where vim has special improved macro-writing capabilities that vi does not.

At the time of the writing of this book, when we did a default installation of our reference Linux systems (Debian-family Debian 9.1, Ubuntu 16.04, Linux Mint 18.2, and CentOS 7.4) they had the following text editors pre-installed:

Debian 9.1: nano, vi

Ubuntu 16.04: nano, vi

Linux Mint 18.2: nano, vi

CentOS 7.4: nano, vi, vim

The above listing is true if you have done a basic installation of the system, as detailed at the beginning of Chapter 17. If someone else, such as a designated system administrator, has done the installation, then you need to confer with them about what editors have been installed. For us, gvim, and GNU emacs were not pre-installed on any of our reference Linux systems.

The easiest and best way to install these editors on your system is by using the package management system available on your system. We show the basics of package management systems for our reference Linux systems in Appendix A of the printed book. We also show how to install vi, vim, and emacs on our reference Linux systems, using the apt and yum commands, in Appendix A. For example, when we show installation of an editor on our Linux Mint system, we searched in the Software Manager for vim, gvim, and emacs, and then used the Linux Mint Software Manager’s facilities to install the following packages-

For vim: Vim Version 2:7.4.1689-3ubuntu1.2

For gvim: Vim-gtk 2:7.4.1689-3ubuntu1.2

For emacs: Gnu emacs 24.5.1

In addition, please be aware that if you are logging into a Linux system via a terminal window (therefore interacting only in a login shell with the system), such as with PuTTY from a Windows machine or a Mac, many of the graphical modes and techniques of using Linux text editors will not be available to you. But that does not prevent you from using the traditional typed commands and keyboard edits that we show for all of the editors! For example, as seen in the printed book Chapter 3, nano uses typed commands exclusively. Another good example is noted below for emacs, where you can exclusively use a non-graphical mode of editing.

W25.1.3 Using Text Editors

Modern Linux uses both a Graphical User Interface (GUI), with powerful window management systems, and a Character User Interface (CUI). Therefore, to do useful things such as execute multiple commands from within a script file, write e-mail messages, or create C language programs, you must be familiar with one or perhaps multiple ways of entering text into a file. In addition, you must also be familiar with how to edit existing text files efficiently—that is, to change their contents or otherwise modify them in some way. Text editors allow you to view a file’s contents, similar to the **more** command, so that you can identify the key features of the file, and then read and utilize the information contained in it. For example, a file without any extension, such as **foo** (rather than **foo.txt**) might be a text file that you can view with a text editor.

The editors that we consider here are all considered full-screen display editors. That is, on the display screen or monitor that you are using to view or edit a file, you are able to see a portion of the file, which fills most or all of the window allocated to the text editor screen display. You are also able to move the cursor, or point, to any of the text you see in this full-screen display, with either the arrow keys on the keyboard or with a mouse. That text material is usually held in a temporary storage area in computer memory called the editor *buffer*. If your file is larger than one screen, the buffer contents change as you move the cursor through the file. The difference between a file, which you edit, and a buffer is crucial. For text-editing purposes, a file is stored on disk as a sequence of data. When you edit that file, you edit a copy that the editor creates, which is in the editor buffer. You make changes to the contents of the buffer—and can even manipulate several buffers at once—but when you save the buffer, you write a new sequence of data to the disk, thereby saving the file.

Another important operational feature of all the editors discussed in this chapter is that, traditionally, their actions are based on keystroke commands, whether they are a single keystroke or combinations of keys pressed simultaneously or sequentially. Because one of the primary input devices in Linux is the keyboard, using the correct syntax of keystroke commands is mandatory. But the keyboard method of input, once you have become accustomed to it, is as efficient or, for some users, even more efficient than mouse/GUI input. Keystrokes also are more flexible, giving you more complete and customizable control over editing actions. Generally, you should choose the editor you are most comfortable with, in terms of the way you prefer to work with the computer. However, your choice of editor also depends on the complexity and quantity of text creation and manipulation that you want to do. Practically speaking, editors such as vi, vim, gvim, and GNU emacs are capable of handling complex editing tasks in multiple windows on multiple files, and provide you with a visual software development environment, as well as document production and management capability. But to take advantage of that power, you have to learn the mechanics of the commands that are needed to perform those tasks and how they are implemented either graphically or by typing them—and retain that knowledge. The basic functions common to the text editors that we cover here are listed in Table W25.1, along with a short description of each function.

|  |  |
| --- | --- |
| **Function** | **Description** |
| Cursor movement | Moving the location of the insertion point or current position in the buffer |
| Cut or copy, paste | “Ripping out” text blocks or duplicating text blocks, reinserting ripped or duplicated blocks |
| Deleting text | Deleting text at a specified location or in a specified range |
| Inserting text | Placing text at a specified location |
| Opening, starting | Opening an existing file for modification, beginning a new file |
| Quitting | Leaving the text editor, with or without saving the work done |
| Saving | Retaining the buffer as a disk file |
| Search, replace | Finding instances of text strings, replacing them with new strings |

Table W25.1 Basic Text-Editing Functions

W25.2 Using the vi, vim, and gvim Editors

The vi, vim, and gvim Linux text editors have almost all the features of a word processor and have tremendous flexibility in creating text files. Initially, they are as easy to use as nano, and can be used to create simple text files for a novice user. Once you begin to exploit their more complex capabilities, their advantages allow you to create, manipulate, and use the kinds of text files that the full range of Linux users, from absolute novice to seasoned veteran, commonly work with. We will proceed in the following section and subsections by demonstrating vi as a text-only interface editor, then move to a more graphical interface approach with vim and gvim.

*Buffers*: As we mentioned in Section W25.1, the notion of a *buffer* as a temporary storage facility for the text that you are editing is very useful and important in vi, vim, and gvim. The main buffer, sometimes referred to as the editing buffer or the work buffer, is the main repository for the body of text that you are trying to create, or to modify from some previous permanently archived file on disk. The general purpose buffer is where your most recent “ripped-out” (cut/copied) text is retained. Indexed buffers allow you to store more than one temporary string of text.

W25.2.1 Basic Shell Script File Creation, Editing, Execution

*Shell Script File*: Practice Session W25.1 shows how to create a script file, or collection of Linux commands that are executed in sequence, and then execute the script. We present more about shell programming and script files in Chapters 12 through 15. For this example, we assume that you are running the Bourne Again (bash) shell. If you are running some other shell by default, go back to Chapter 2, Section 2.8, and review how to identify and change shells.

In Linux , do Practice Session W25.1 for the bash shell, which is the default shell in that system, and don’t change shells.

And do not worry too much if you make an error in Steps 2, 3, and 4; you can go through the rest of the script file discussion and then come back to this example after you have learned some of the editing commands and become more familiar with them.

Practice Session W25.1

Step 1: At the shell prompt, start vi by typing **vi firscrip** and then pressing **<Enter>**. The vi screen appears on your display.

Step 2: Type **A**. Then type **ls -la** and then press **<Enter>**.

Step 3: Type **who** and then press **<Enter>**.

Step 4: Type **pwd** and then press the **<Esc>** key. At this point, your screen should look like that shown in Figure W25.1.

Step 5: Type **:wq** and then press **<Enter>**.

Step 6: At the shell prompt, type **bash firscrip** and then press **<Enter>**.

Step 7: Note the results. How many files do you have in your present working directory? What are their names and sizes? Who else is using your computer system? What is your present working directory?

ls -la

who

pwd

~

Figure W25.1 File firscrip after Step 4.

In Practice Session W25.1, you accomplished these things:

\* At Step 2, typing A took vi out of *Command mode* (which is what vi starts in by default) and placed it in one of the forms of Insert mode. In other words, anything that you typed at the keyboard was appended as text on the first line in the text area of the editor.

\* When you pressed the <Esc> key in Step 4, vi was taken out of Insert mode and put back into Command mode.

\* When you typed : in Step 5, that was a valid Command mode prefix character for the two commands that followed, and put vi in *Last Line mode*.

\* When you typed wq after the :, vi interpreted those commands in Last Line mode as write out or save the file, and quit the editor.

\* Step 6 executes the Bash shell script file.

In-Chapter Exercises

1 Launch vi in a terminal, or console, window on your Linux system, by just typing vi on the command line. Then create a simple text file, and save it as firstext.txt in your current working directory. Finally, gracefully exit the vi editor. What commands did you use to accomplish everything you did for this exercise?

2. Launch vi in a terminal, or console, window on your Linux system, by just typing vi on the command line. Then create another bash script file, similar to the one shown in Practice Session W25.1, but placing other common single Linux commands in it. Save it as 2ndscrip in your current working directory. Finally, gracefully exit the vi editor, and test this script file on the bash command line. What commands did you use to accomplish everything you did for this exercise?

W25.2.2 How to Start, Save a File, and Exit

When you need to do Linux text editing that gives you as much functionality as a typical word processor, you can use the vi text editor. To start vi from the command line, use the following general syntax (anything enclosed in square brackets [ ] is optional):

**vi [options] [file(s)]**

**Purpose:** Allows you to edit a new or existing text file(s)

**Output:** With no options or file(s) specified, you are placed in the vi program and can begin to edit a new buffer

**Commonly used options/features:**

**+n** Begin to edit file(s) starting at line number n

**+/exp** Begin to edit at the first line in the file matching string exp

The operations that you perform in vi fall into two general categories: *Command mode* operations, which consist of key sequences that are commands to the editor to take certain actions, and *Insert mode* operations, which allow you to input text.

The general organization of the vi text editor and how to start, exit, and switch modes are illustrated in Figure W25.2. The general organization of vim and gvim, and how to start, exit, and switch modes in those editors, is the same as shown for vi in Figure W25.2.

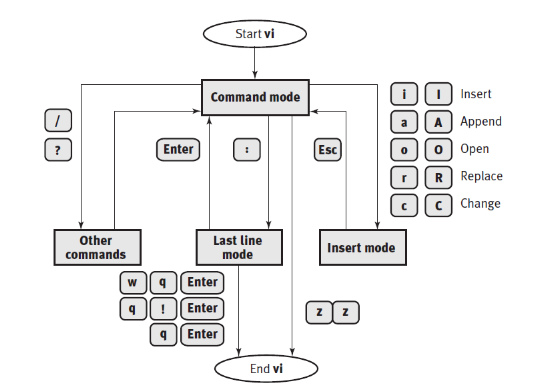


Figure W25.2 General Organization of vi, vim, and gvim

For example, to change from Command mode, which you are in when you first enter the editor, to Insert mode, type a valid command, such as A to append text at the end of the current line. Certain commands that are prefixed with the :, /, ?, or :! characters are echoed or shown to you on the last line on the screen and must be terminated by pressing <Enter>. *Last Line mode*, sometimes called *ex mode* because it is derived from the ex editor, allows you to execute certain commands and leave the editor. To change from Insert mode to Command mode, press the <Esc> key.

The keystroke commands that you execute in vi are case sensitive; for example, uppercase A appends new text after the last character at the end of the current line, whereas lowercase a appends new text after the character the cursor is on.

To start vi, at the shell prompt, type vi (and optionally designate some option[s] and file name[s]) and then press <Enter>. You are now in Command mode. To enter Insert mode, type A and you are now able to insert text on the first line of the file.

After entering text, you can press the <Esc> key to enter Command mode.

At any point in your creation or manipulation of text, you can press the u key on the keyboard to undo the last operation.

From Command mode, you can save the text that you just inserted into the buffer to a file on disk by typing :w filename and pressing <Enter>, where filename is the name of the file you want to save the text to. To quit the editor, type :q.

W25.2.3 The Format of a Vi Command and the Modes of Operation

In Command mode, the generic syntax of keystrokes is:

**[#1] operation [#2] target**

where:

anything enclosed in **[ ]** is optional;

**#1** is an optional number, such as 5, specifying how many operations are to be done;

**operation** is what you want to accomplish, such as deleting lines of text;

**#2** is an optional number, such as 5, specifying how many targets are affected by the **operation**; and

**target** is the text that you want to do the operation on, such as an entire line of text.

Note that if the current line is the target of the operation, the syntax for specifying the target is the same as the syntax of the operation; for example, dd deletes the current line. Also, a variation on this generic syntax is the cursor movement command, whereby you can omit the numbers and operation and simply move the cursor by word, sentence, paragraph, or section. Table W25.2 lists some specific examples of this generic syntax and variations used in Command mode.

|  |  |
| --- | --- |
| **Command** | **Action** |
| cw | Change word. |
| cc | Change line. |
| c$ | Change text from current position to end of line. |
| C | Same as c$. |
| dd | Delete current line. |
| 7 dd | Delete 7 lines. |
| d$ | Delete text from current position to end of line. |
| D | Same as d$. |
| 5dw | Delete 5 words. |
| d7,14 | Delete lines 7 through 14 in the buffer. |
| d} | Delete up to next paragraph. |
| d^ | Delete back to beginning of line. |
| d/ pat | Delete up to first occurrence of pattern. |
| dn | Delete up to next occurrence of pattern. |
| df x | Delete up to and including x on current line. |
| dt x | Delete up to (but not including) x on current line. |
| dL | Delete up to last line on screen. |
| dG | Delete to end of file. |
| gqap | Reformat current paragraph to text width (vim and gvim). |
| g~w | Switch case of word (vim and gvim). |
| guw | Change word to lowercase (vim and gvim). |
| gUw | Change word to uppercase (vim and gvim). |
| p | Insert last deleted or yanked text after cursor. |
| gp | Same as p, but leave cursor at end of inserted text (vim and gvim). |
| gP | Same as P, but leave cursor at end of inserted text (vim and gvim). |
| ]p | Same as p, but match current indention (vim and gvim). |
| [p | Same as P, but match current indention (vim and gvim). |
| P | Insert last deleted or yanked text before cursor. |
| r x | Replace character with x. Does not require the use of <Esc>! |
| R text | Replace with new text (overwrite), beginning at cursor. <Esc> ends replace mode. |
| s | Substitute character. <Esc> ends substitute mode. |
| 4s | Substitute four characters. <Esc> ends substitute mode. |
| S | Substitute entire line. <Esc> ends substitute mode. |
| u | Undo last change. |
| <Ctrl+R> | Redo last change (vim and gvim). |
| U | Restore the current line, if you have not moved off of it. |
| x | Delete current cursor position. |
| X | Delete back one character. |
| 5X | Delete previous 5 characters |
| . | Repeat last change. |
| ~ | Change case and move cursor right. |
| <Ctrl+A> | Increment number at the cursor (vim and gvim). |
| <Ctrl+X> | Decrement number at the cursor (vim and gvim). |

Table W25.2 Examples of vi Command Syntax

As previously stated, when you start vi, it is in Command mode. When you want to be in Insert mode instead of Command mode, press a valid key to accomplish the change. Some of these keys are shown in Table W25.3.

|  |  |
| --- | --- |
| **Key** | **Action** |
| a | Appends text after the character the cursor is on |
| A | Appends text after the last character of the current line |
| c | Begins a change operation, allowing you to modify text |
| C | Changes from the cursor position to the end of the current line |
| i | Inserts text before the character the cursor is on |
| I | Inserts text at the beginning of the current line |
| o | Opens a blank line below the current line and puts the cursor on that line |
| O | Opens a blank line above the current line and puts the cursor on that line |
| R | Begins overwriting text |
| s | Substitutes single characters |
| S | Substitutes whole lines |

Table W25.3 Important Keys Used to Switch From Command to Insert Mode

After inserting text, you can edit the text, move the cursor to a new position in the buffer, and save the buffer and exit the editor—all from within Command mode. When you want to change from Insert mode to Command mode, press the <Esc> key.

To save the buffer and exit the editor, press the : key (colon) to enter Last Line mode. The general commands that are useful in Last Line mode are shown in Table W25.4.

|  |  |
| --- | --- |
| **Command** | **Action** |
| : n, m w file | Write lines n to m to new file. |
| : n, m w >> file | Append lines n to m to existing file. |
| :r filename | Reads and inserts the contents of the file filename at the current cursor position |
| :wq | Saves the buffer and quits |
| :w | Saves the current buffer and remains in the editor. |
| :w filename | Saves the current buffer to filename |
| :w! filename | Overwrites filename with the current text |
| :w! | Write file (overriding protection). |
| :w! file | Overwrite file with current text. |
| :w %.new | Write current buffer named **file** as **file.new**. |
| :q | Quit vi (fails if changes were made). |
| :q! | Quit vi without saving the buffer. |
| :Q | Quit vi and invoke ex. |
| :vi | Return to vi after Q command. |
| ZZ | Quits vi, saving the file only if changes were made since the last save |
| % | Replaced with current filename in editing commands. |
| # | Replaced with alternate filename in editing commands. |

Table W25.4 Important Commands in Command Mode

For now, we recommend that you use the arrow keys on the keyboard to move the cursor around in the buffer. It is possible to also use the h, j, k, and l keys on the keyboard to move the cursor. In gvim, you can use the mouse and its buttons!

The following practice session introduces you to some of the commands presented in Tables W25.2 through W25.4. Feel free to use either vi or vim to do the steps of the following on your Linux system:

Practice Session W25.2

Step 1: At the shell prompt, type **vi firstvi** and then press <Enter>.

Step 2: Type A, then type **This is the first line of a vi file.** and then press **<Enter>**.

Step 3: Type **This is the line of a vi file.** and then press **<Enter>**.

Step 4: Type **is the 3r line of a vi**

Step 5: Press the **<Esc>** key.

Step 6: Type **:w** and then press **<Enter>**.

Step 7: Use the arrow keys on the keyboard to position the cursor ob the character l in the word line on the second line of the file.

Step 8: Type **i** and then **2nd\_** , where the \_ is a space character

Step 9: Press the **<Esc>** key.

Step 10: Use the arrow keys to position the cursor anywhere on the third line of the file.

Step 11: Type **I** and then **This\_** , where the \_ is a space character.

Step 12: Press the **<Esc>** key.

Step 13: Use the arrow keys on the keyboard to position the cursor on the character r in 3r on this line.

Step 14: Type **a** and then **d**.

Step 15: Press the **<Esc>** key.

Step 16: Type **A** and then **file**.

Step 17: Press the **<Esc>** key on the keyboard. Your screen display should look similar to Figure W25.3.

Step 18: Type **:wq**. You will be back at the shell prompt.

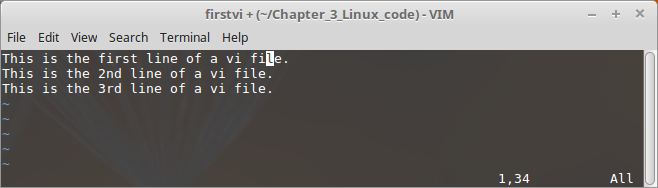


Figure W25.3 File firstvi

The following in-chapter exercise asks you to apply some of the operations you learned about in the previous practice session.

In-Chapter Exercises

3. With vi you begin editing a file that you created yesterday. You want to save a copy of it with a different filename while still in vi, but you don’t want to quit this editing session. How do you accomplish this result in vi?

4. What happens if you accomplish five operations in vi and then type 5u when in Command mode?

5. In your own words, give a brief description of what the following vi commands accomplish. Use Tables W25.1 through W25.5 to guide you in your answers.

dd, d1,12, x, a, 5b, 3w, :q!, G, 1G .

W25.2.4 Cursor Movement and Editing Commands

In Command mode, several commands accomplish cursor movement and text editing tasks. Table W25.5 lists important cursor movement and keyboard editing commands. As we have already noted, character-at-a-time or line-at-a-time moves of the cursor can be accomplished easily with the arrow keys, or alternatively with the h, j, k, and l keys on the keyboard.

The following practice session lets you continue editing the file you created in Practice Session W25.2 by using commands presented in Table W25.5.

|  |  |
| --- | --- |
| **Command** | **Action** |
| 1G | Moves the cursor to the first line of the file |
| G | Moves the cursor to the last line of the file |
| 0 (zero) | Moves the cursor to the first character of the current line |
| <Ctrl+G> | Reports the position of the cursor in terms of line # and column # |
| $ | Moves the cursor to the last character of the current line |
| w | Moves the cursor forward one word at a time |
| b | Moves the cursor backward one word at a time |
| x | Deletes the character at the cursor position |
| dd | Deletes the line at the current cursor position |
| u | Undoes the most recent change |
| r | Replaces the character at the current cursor location with what is typed next |

Table W25.5 Cursor Movement and Keyboard Editing

Practice Session W25.3

Step 1: At the shell prompt, type **vi firstvi** and then press **<Enter>**.

Step 2: Type **G**. The cursor moves to the last line of the file.

Step 3: Hold down the **<Ctrl>** and **g** keys at the same time. On the last line of the screen display, vi reports the following:

"firstvi" 3 lines --100%-- 3,1 All

This is a report of the buffer that you are editing, the total number of lines in the buffer, the percentage of the buffer that this line represents, the current column position of the cursor, and other editor settings that are the default.

Step 4: Type **o**. A new line opens below the third line of the file

Step 5: Type **This is the 5th line of a vi file.** Type **<Esc>**.

Step 6: Type **0** (zero). The cursor moves to the first character of the line you just typed in.

Step 7: Type **$**. The cursor moves to the last character of the current line.

Step 8: Type **O**. A new line opens above the current fourth line.

Step 9: Type **This is the 44th line of a va file.** Type **<Esc>**.

Step 10: Use the arrow keys to position the cursor over the first 4 in 44 on this line.

Step 11: Type **x**.

Step 12: Use the arrow keys to position the cursor over the a in va on this line.

Step 13: Type **r** and then type **i**.

Step 14: Type **dd**.

Step 15: Type **:wq** to go back to the shell prompt.

Step 16: At the shell prompt, type **more firstvi** and then press **<Enter>**. How many lines with text on them does **more** show in this file?

W25.2.5 Yank and Put (Copy and Paste) and Substitute (Search and Replace)

Every word processor is capable of copying and pasting text and also of searching for old text and replacing it with new text. Copying and pasting are accomplished with the vi commands yank and put. In general, you use yank and put in sequence and move the cursor (with any of the cursor movement commands or methods) only between yanking and putting. Some examples of the syntax for yank and put are given in Table W25.6.

|  |  |
| --- | --- |
| **Command Syntax** | **What It Accomplishes** |
| y2W | Yanks two words, starting at the current cursor position, going to the right |
| 4yb | Yanks four words, starting at the current cursor position, going to the left |
| yy or Y | Yanks the current line |
| p | Puts the yanked text after the current cursor position |
| P | Puts the yanked text before the current cursor position |
| 5p | Puts the yanked text in the buffer five times after the current cursor position |
| Y | Copy current line |
| yy | Copy current line |
| " x yy | Copy current line to register x |
| ye | Copy text to end of word |
| yw | Like ye, but include the whitespace after the word |
| y$ | Copy rest of line |
| " x dd | Delete current line into register x |
| " x d | Delete into register x |
| " x p | Put contents of register x |
| y]] | Copy up to next section heading |
| J | Join current line to next line |
| gJ | Same as J, but without inserting a space (vim and gvim) |
| :j | Same as J |
| :j! | Same as gJ |

Table W25.6 Example Yank and Put Command Syntax

The simple vi forms of search and replace are accomplished using the substitute command. This command is executed when vi is in Last Line mode, where you preface the command with the : character and terminate the command by pressing <Enter>. The format of the substitute command as it is typed on the status line is:

**:[range]s/pattern/string[/option(s)][count]**

where:

anything enclosed in **[ ]** is not mandatory;

**:** is the colon prefix for the Last Line mode command;

**range** is a valid specification of lines in the buffer (or the current line is the **range**);

**s** or **substitute** is the syntax of the substitute command;

**/** is a delimiter for searching;

**pattern** is the text or objects you want to replace;

**/** is a delimiter for replacement;

**string** is the new text or objects;

**/option(s)** is a modifier, usually **g** for global, to the command; and

**count** is the number of lines to execute the command on from the current position.

The grammar of pattern and string can be extremely explicit and complex, and may take the form of a regular expression. (We present more information on the formation of regular expressions in Chapter 7, Section 7.2.) Some examples of the syntax for the substitute command, including vim/gvim-only constructions, are given in Table W25.7.

|  |  |
| --- | --- |
| **Command Syntax** | **What It Accomplishes** |
| :s/john/jane/ | Substitutes the word jane for the word john on the current line, only once. |
| :s/john/jane/g | Substitutes the word jane for every word john on the current line. |
| :1,10s/big/small/g | Substitutes the word small for every word big on lines 1–10. |
| :1,$s/men/women/g | Substitutes the word women for every word men in the entire file. |
| :'<,'>s/this/that/g | Select the range in Command mode first by typing <Ctrl+V> and using the arrow keys. Then type **:**. The word that will be substituted for the word this (vim, gvim only). |
| :s/ \<tim\>/tom/ | Substitutes only the whole word tim with the word tom, not the partial match of tim in any string. |
| :%s/terrible/wonderful/gc | Interactive substitution using c option of the word terrible with the word wonderful (vim, gvim only). |
| :%s/^/ \=line(".") . ". "/g | Makes the line numbers of all lines in the buffer permanently part of each line (vim, gvim only). |

Table W25.7 Example Syntax for the Substitute Command

Practice Session W25.4 shows you how to use the vi commands yank and put to copy and paste. It also allows you to do individual and multiple searches and replace text with the vi substitute command.

Practice Session W25.4

Step 1: At the shell prompt, type **vi multiline** and then press **<Enter>**.

Step 2: Type **A** and then type **Windows is the operating system of choice for everyone.**

Step 3: Press the **<Esc>** key. You have left Insert mode and are now in Command mode.

Step 4: Press the **0** (zero) key. The cursor moves to the first character of the first line.

Step 5: Type **yy**. This action yanks, or copies, the first line to a special buffer.

Step 6: Type **7p**. This action puts, or pastes, the first line seven times, creating seven new lines of text containing the same text as the first line. The cursor should now be on the first character of the eighth line. If it’s not, use the arrow keys to put the cursor there.

Step 7: Type **1G**. This action puts the cursor on the first character of the first line in the buffer.

Step 8: Hold down the **<Shift>** and **;** keys at the same time. Doing so places a **:** in the status line at the bottom of the vi screen display, allowing you to type a command.

Step 9: Type **s/everyone/students/** and then press **<Enter>**. The word everyone at the end of the first line is replaced with the word students.

Step 10: Use the arrow keys to position the cursor on the first character of the second line.

Step 11: Type **:s/everyone/computer scientists/** and then press **<Enter>**.

Step 12: Repeat Steps 8–10 on the second through eighth lines of the buffer, substituting the words engineers, system administrators, web servers, scientists, networking, nerds, and mathematicians for the word everyone on each of those seven lines. Type **1G**

Step 13: Type **:1,$s/Windows/Linux /g** and then press **<Enter>**. You have globally replaced the word Windows on all eight lines of the file with the word Linux . Correct?

Step 14: Type **:wq**. You have now saved the changes and exited from vi.

W25.2.6 Vim and Gvim

Vim and gvim are two examples among many of enhanced, “improved” versions of vi. The following subsections illustrate some of the advantages of using vim and gvim over the traditional vi editor.

*W25.2.6.1 Vim Enhancements*

The following capabilities of vim that enhance vi functionality, particularly the first one shown, are suggestions that you can use to expedite your editing tasks with vim and gvim over and above the capabilities of vi:

\* vimrc

If you want to enable any of the improved facilities of vim and gvim, you should create a **~/.vimrc** file. Even if this file is empty, it will enable the facilities that we illustrate in this section!

\* Help

In vim Last Line mode, type help or press the <F1> function key.

Vim opens a help buffer that gives you extensive help on its facilities. In Last Line mode, when in the help buffer, type q to exit help.

\* Multiple Windows

The Last Line mode command split splits the current window in two. You can then move the cursor up to a window with <Ctrl+W> j and down a window with <Ctrl+W> k. For example, the Last Line mode command split new.c splits the window and begins editing the file named **new.c**. To close a window, use the normal vim exit commands ZZ or :q!.

\* Multiple Levels of Undo

Unlike vi, you can use the undo command to undo several steps back in the command history. For example, typing u in Command mode undoes the last action in vim, and typing 3u in Command mode undoes the last three actions you did in vim. The undo level is set by default to 1000. You can redo multiply as well, using <Ctrl+R>. For example, 3 <Ctrl+R> redoes the last 3 actions that were undone with u.

\* Visual Mode

Typing v causes vim to enter *Visual mode*. You can then highlight a block of text and execute a vim Command mode operation on it. The v command selects text by character. The <Ctrl+V> command selects text as a block. The V command selects the current line. See Section W25.2.6.2 for more details on this facility in vim.

\* The incsearch and hlsearch Environmental Options (Incremental Search and Highlight Search)

For the incremental search, by default, searching starts after you enter the string. With the option:

**:set incsearch**

incremental searches will be done. The vim editor will start searching when you type the first character of the search string. As you type in more characters, the search is refined.

For the highlight search option, setting the option turns on search highlighting. This option is enabled by the command:

**:set hlsearch**

After the option is enabled, any search highlights the string matched by the search.

\* The cindent Environmental Option and the = Command Option

Like vi autoindent, the vim editor does a more specific form of indentation. The cindent option is set with the command:

**:set cindent**

This turns on C style indentation. Each new line will be automatically indented the correct amount according to the C indentation standard.

\* The :make Command

To compile a C program with an accompanying make file, and correct the errors, you can type this command in Last Line mode:

**make**

This runs the make command and captures the output. When the command finishes the editor starts editing the first file. The next step is to fix the error. After that you need to go to the line causing the next error. This is done using the command:

**cn**

This command will go to the location of the next error even if it is in another file.

You can continue fixing problems and using cn until all your problems are over or you want to do a recompile. If you want to see the current error message again, use the command:

cc

\* Last Line Mode Command History

When you are in Last Line mode, you can use the <Up> arrow key to recall an older command line entry, and then can use the <Down> arrow key to go forward to newer commands. Then, when you press <Enter> after you have indexed to that previous command in the history, that previous command is executed again.

There are four histories you can utilize in vim, but the two most important ones are for:

* Last Line mode command history
* / and ? search command history

Your search history is most useful to you, particularly because if you type complex search criteria, you do not want to have to retype them every time you want to repeat that search!

The two other histories are for expressions and input lines for the input() function.

As an example, you have done a Last Line mode command, typed five more Last Line mode commands, and then want to repeat the first command again. To do this, in Last Line mode press the <Up> arrow key five times. Another way of doing this is to type the first few letters of the Last Line mode command you want to return to.

The <Up> key will use the text typed so far and compare it with the lines in the history. Only matching lines will be used.

If you do not find the line you were looking for, use the <Down> arrow key to go back to what you typed and correct that. You can also type <Ctrl+U> to start all over again.

To see all the lines in your Last Line mode command history, while in Last Line mode, type:

**history**

You will then see a complete history of the Last Line mode commands for this session at the bottom of the screen display.

Your entire search history for this session is displayed by typing history/ in Last Line mode.

<Ctrl+P> will work like the <Up> arrow key, except that it doesn’t matter what you already typed. <Ctrl+N> works like the <Down> arrow key.

\* The Last Line Mode Command Line Window

Typing any text in the Last Line mode command history to modify a previous command and then execute it is possible but difficult.

A better way to use a modified form of a Last Line mode command from the history is to open the *command line window* while in Command mode by typing:

**q:**

Vim now opens a small utility window at the bottom of the screen. It contains the command line history and an empty line at the end, similar to this illustration:

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

|other window |

|~ |

|first.c=============================|

|:w second.c |

|:w third.c |

|:w fourth.c |

|:w fifth.c |

|:w sixth.c |

|:history

|: |

|command−line=========================|

| |

~/project.c

+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−+

In the buffer in this small utility window, you are in Insert mode, and can use Insert mode commands to modify text and also move commands. You can use the arrow keys to move around.

For example, move up the history tree with the <Up> arrow key to the :w third.c line.

Change the word third to thirteenth.

Now press <Enter> when on that line, and this command will be executed. The command line window will then close. The <Enter> command will execute the line under the cursor. This works if vim is in Insert mode or in Command mode.

Unfortunately, changes you make in the command line window are lost! They do not result in any changes in the command history itself, but the command you execute when you are in the command line window will be added at the end of the history, similar to all other executed commands. Also, only one command line window can be open at a time.

The command line window is very useful when you want to see your old command history, index to a particular command, edit it, and execute it.

A search command in your history can be used to find something new if you index to it and modify it. For example, if in the command line window one of the lines contained :s/everyone/computer scientists/, you could index to it in the command line window and modify and execute it.

\* Word Completion

When you are typing and you enter a partial word, you can cause vim to search for a completion by using <Ctrl+P> (search for previous matching word) and <Ctrl+N> (search for next match).

\* Record and Playback

The . (period) command repeats the previous change in Command mode. To accomplish multiple, complex changes in vim Command mode, you can use the *record and playback* facility. There are three steps in record and playback:

1. The q(register) command starts recording keystrokes into the key named register. The register name must be a letter of the alphabet.
2. Type the commands you want to record in the register.
3. To end recording, press q.

You can now execute the macro by typing the command @register. For example, you have a list of filenames in a buffer that looks like this:

stdio.h

fcntl.h

unistd.h

stdlib.h

And what you want is the following:

#include "stdio.h"

#include "fcntl.h"

#include "unistd.h"

#include "stdlib.h"

You start by moving to the first character of the first line. Next, in Command mode, you execute the following commands:

**qa**

**^**

**i#include "<Esc>**

**$**

**a"<Esc>**

**j**

**q**

These commands do the following:

1. Start recording a macro in register a.
2. Move to the beginning of the line.
3. Insert the string #include " at the beginning of the line.
4. Move to the end of the line.
5. Append the double quotation mark (") character to the end of the line.
6. Go to the next line.
7. Stop recording the macro.

Now that you have done the work once, you can repeat the change by typing the command "@a" three times.

The "@a" command can be preceded by a count, which will cause the macro to be executed that number of times. In this case you would type: "3@a".

In-Chapter Exercises

6. How would you open a unique history window for the / and ? commands?

7. Where does the cursor have to be positioned in the buffer if you want to execute a modified version of the substitute command :s/everyone/computer scientists/ correctly?

8. Can you include Last Line mode commands, such as substitute, or write to a file, in a record and playback session?

*W25.2.6.2 Vim Visual Mode*

Because vi does not have a graphical, or “visual,” method of selecting and operating on blocks of text, we use vim Visual mode. In vim, Visual mode is the graphical and easy way to select a block of text in order to use a prescribed operator on it. The following will briefly describe Visual mode’s features and give a simple example. Vim Visual mode allows you to apply commands to blocks of text that can be selected graphically, even though you may not be in a GUI environment. In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

Using Visual mode is done in three steps:

Step 1: Move the cursor to the start of the text block, mark the start of the block with "v" (character mode), "V" (line mode), or <Ctrl+V> (blockwise mode). The character under the cursor will be used as the start of the block.

Step 2: Depending on what kind of functionality is provided in the terminal or console window you are working in, move to the end of the text block, either with the arrow keys on the keyboard, with the h, j, k, or l keys on the keyboard, or with the mouse and mouse button(s). The text from the character where you start Visual mode, up to and including the character under the cursor, is highlighted. Generaly v and V modes allow definition of nonrectangular blocks, whereas <Ctrl+V> allows definition of only rectangular blocks.

Step 3: Type a prescribed operator command. The highlighted characters from Step 2 will be operated upon depending on the nature of the prescribed operator listed.

You can use <Esc> to stop the definition of a block any time before you use a prescribed operator.

A simple example that illustrates how you can copy and paste using Visual mode follows:

Step 1: At the shell prompt type **vim visualtest1**, then press **<Enter>** on the keyboard.

Step 2: Type three or four arbitrary lines of text of uneven length (five to ten words each) into the buffer that opens on screen. Put some spaces at the beginning of some of the lines.

Step 3: Position the cursor on the first character of the first line of the buffer.

Step 4: Type **v**. On the last line display you will be notified that you have entered Visual mode! Now you can define the block that will be all or possibly only a portion of Step 2 text.

Step 5: Expand the highlighted area by using the input device of your choice until all the text you typed in Step 2 is highlighted. On our display, we used the arrow keys, and the highlighted area shows greyed-out (in fact, that’s how we knew we were defining the area to be yanked.)

If you make a mistake in defining the block, use **<Esc>** to stop the block definition and begin highlighting again at the first character in the buffer until you get the block definition you desire.

Step 6: Type **y**.

Step 7: On the last line display you will see a report of how many lines you just yanked.

Step 8: Position the cursor anywhere on the last line of the buffer.

Step 9. Type **o**. A new line opens below the last line in the buffer. Press **<Esc>**.

Step 10: Type **p**. The yanked block from Step 6 is put back in the buffer, starting on the new line you opened in Step 9 and proceeding downward. Save the file if you want to.

*W25.2.6.3 Using Gvim to Cut and Paste between Multiple Open Buffers*

To illustrate the speed and efficiency of using gvim as a modern graphical Linux text editor, and to describe some of gvim’s functions, the following practice session allows you use gvim to create text in two different files, open buffers into those files in two different windows, and copy and paste between those buffers.

In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

Practice Session W25.5

Step 1: At the shell prompt, type **gvim gvim1** and then press the **<Enter>** key.

Step 2: Type **A** and then type **This is the first line of text.** Then press the **<Enter>** key twice.

Step 3: Type **This is the third line of text.** Then press the **<Enter>** key.

Step 4: From the gvim pull-down menu, make the choice **Window>Split**. You now are looking into two windows on the same buffer.

Step 5: From the gvim pull-down menu, make the choice **File>Save**. Click the **OK** button in the Save window, if one appears. The buffer is saved to the file **gvim1**.

Step 6: Use the mouse and click anywhere in the lower window with the left mouse button. You are now working in the lower buffer.

Step 7: From the gvim pull-down menu, make the choice **File>Save As**. In the Name: box, change the name of the file to **gvim2**, and then make the **Save** button choice. The buffer is saved as **gvim2**, and you are looking into the buffer through two windows.

Step 8: The active buffer is still seen in the lower window. Use the mouse and **<Delete>** key on the keyboard to change the word first to the word second, and the word third to the word fourth in the lower window.

Step 9: Click anywhere in the top window.

Step 10: Make the gvim pull-down menu choice **File>Open**. Scroll down and open **gvim1** in the current directory by selecting it and making the **OK** button choice. You should now be seeing **gvim1** in the upper window, and **gvim2** in the lower window.

Step 11: Click anywhere in the bottom window.

Step 12: Use the mouse and left mouse button to highlight the text This is the second line of text.Make sure the cursor is flashing on the period as you finish selecting that line.

Step 13: Make the gvim pull-down menu choice **Edit>Copy**. You have “yanked” a line of text in the lower buffer graphically.

Step 14: Click on the second blank line in the upper window buffer.

Step 15: Make the gvim pull-down menu choice **Edit>Paste**. The line This is the second line of text. is now on the second line of the upper window buffer. You can use the gvim pull-down menu choice **Edit>Undo** to correct mistakes in copying and in pasting.

Step 16: Repeat Steps 11 through 15 to copy and paste the line This is the fourth line of text. from the lower window buffer to the upper window buffer. When you are done, your screen display should look similar to Figure W25.4.

Step 17: While the active window buffer is the upper window, make the gvim pull-down menu choice **File>Save-Exit**.

Step 18: At the shell prompt, type **more gvim1**. What appears on screen? Do the same for the file **gvim2**. What appears on screen?

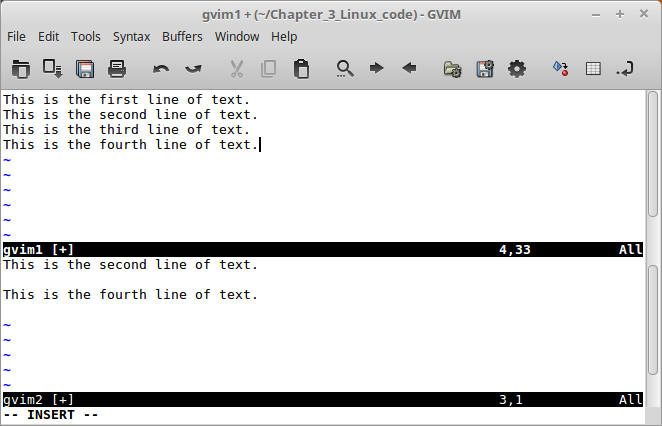


Figure W25.4 File gvim1 After Step 16.

A complete summary of vi, vim, and gvim commands is given in Table W25.8, which we conveniently append to the end of this chapter so that you may print it for use as a handy paper reference when doing text editing with these editors.

In-Chapter Exercises

9. Repeat Practice Session W25.5 for gvim using another text file with content of your own choosing, and then cut and paste from it into 3 other files that you open simultaneously in the same gvim terminal window. Save all 4 files when finished.

10. Similar to Practice Session W25.5, use non-graphical vi in a terminal window on a non-GUI-based Linux system to achieve the same results. What commands did you use to do this? Which editor was easier to use to achieve the same results?

W25.2.7 Changing Vi, Vim, and Gvim Behavior

In general, all of the environment options commands shown in this section work in vi, vim, and gvim. Note that, because vim stands for *vi improved*, vim and gvim have many more environmental options. As previously suggested, you can create an empty version of the **~/.vimrc** file to enable many of the behavioral changes we show here. We also suggest you modify both your **~/.exrc** and **~/vimrc** files to accomplish the behavioral changes illustrated, depending on which editor you want the changes to be implemented in. If you put a *vim-specific behavior-changing option* in the **.exrc** file, when you run vi, you will probably get a warning message in vi, but not a fatal error message.

You can modify any of several environment options to customize the behavior of the vi, vim, and gvim editors, either when you are in the editor at a given time, or for every editor session. These options include, for example, specifying maximum line length and automatically wrapping the cursor to the next line, displaying line numbers as you edit a file, and displaying the mode that the editor is in. You can use full or abbreviated names for most of the options. Some of the most important and useful options and their abbreviations are summarized in Table W25.8a. Also see Table W25.8b for a summary of the use of the set command.

|  |  |  |
| --- | --- | --- |
| **Option** | **Abbreviation** | **Purpose** |
| autoindent | ai | Aligns the new line with the beginning of the previous line. |
| ignorecase | ic | Ignores the case of a letter during the search process (with a / or the ? command). |
| list | list | Displays invisible characters, such as ^I for <Tab> and a $ for end-of-line characters. |
| nolist | nolist | Turns off the display of invisible characters. |
| noignorecase | noic | Instructs cases to be case sensitive. |
| number | nu | Displays line numbers when a file is being edited; line numbers are not saved as part of the file. |
| nonumber | nonu | Hides line numbers. |
| scroll |  | Sets the number of lines to scroll when the <Ctrl+D> command is used to scroll the vi screen up. |
| set |  | Displays all the vi variables that are set. |
| all |  | Displays all set vi variables and their current values. |
| showmode | smd | Displays the current vi mode in the bottom right corner of the screen. |
| noshowmode | nosmd | Turns off the mode of operation display. |
| wrapmargin | wm | Sets the wrap margin in terms of the number of characters from the end of the line, assuming a line length of 80 characters. |

Table W25.8a Important Environmental Options for vi,vim, and gvim

|  |  |
| --- | --- |
| **Last line mode syntax** | **What it does** |
| **abbr command** |  |
| :ab in out | Use in as abbreviation for out in Insert mode. |
| :unab in | Remove abbreviation for in. |
| :ab | List abbreviations. |
| **map!, map commands** |  |
| :map string sequence | Map characters string as sequence of commands. Use #1, #2, etc., for the function keys. |
| :unmap string | Remove map for characters string. |
| :map | List character strings that are mapped. |
| :map! string sequence | Map characters string to input mode sequence. |
| :unmap! string | Remove input mode map (you may need to quote the characters with <Ctrl+V>). |
| :map! | List character strings that are mapped for input mode. |
| qx | Record typed characters into register specified by letter x (vim and gvim). |
| q | Stop recording (vim and gvim). |
| @x | Execute the register specified by letter x. Use @@ to repeat the last @ command. |
| **set command** |  |
| :set x | Enable boolean option x, show value of other options. |
| :set nox | Disable option x. |
| :set x=value | Give value to option x. |
| :set | Show changed options. |
| :set all | Show all options. |
| :set x? | Show value of option x. |

Table W25.8b Last Line Mode Syntax

The set command in Last Line mode changes environmental options. There are two types of environmental options that can be modified with the set command: toggle options, which are either “on” or “off,” and options that require the use of an argument.

For example, after typing :set showmode, you have toggled the mode display “on,” and the editor displays the current mode at the bottom of the screen. If you then type :set noshowmode, you have toggled the mode display “off.” Similarly, after typing :set nu, vi displays the line numbers for all the lines in the file. To turn “off” the line number display, type :set nonu. When the :set ai command has been executed, the next line is aligned with the beginning of the previous line. This useful feature allows you to easily indent source codes that you compose with vi. Pressing <Ctrl+D> on a new line moves the cursor to the previous indentation level.

To see a listing of what all environment options in the editor are (the ones you have modified and the defaults) at any time, type :set all.

To see a listing of what environment options you have modified, either for this session only, or for all sessions, type :set.

When you use set to modify the environment options within an editor session, the options are set for that session only!

If you want to customize your environmental options for all vi, vim, and gvim sessions, you need to put your options in the **.exrc** file in your home directory. You can use the set command to modify one or more options in the **.exrc** file as follows (typing the two keyboard keys <Ctrl+C> terminates the creation of the cat command):

$ **cat > .exrc**

**set wm=5 showmode nu ic**

**<Ctrl+C>**

$

The wm=5 option sets the wrap margin to 5, and is an example of a set command that requires an argument. That is, each line will be up to 75 characters long. The ic option allows you to search for strings without regard to the case of a character. Thus, after this option has been set, the /Hello/ command searches for strings hello and Hello.

**Exercise W25.6**

After examining Table W25.8a and W25.8b, select a few of the environment options that most appeal to you and then place them in your **~/.exrc file**. Test them by running vi.

**Exercise W25.7**

If you haven’t already done so, place the set showmode environment setting in your **~/.exrc** and in your **~/.vimrc** file. Run vim and then gvim. Do various operations in both those editors. Does the mode you are in appear in the mode line in both editors?

W25.2.8 Executing Shell Commands from Within Vi, Vim, and Gvim

At times you will want to execute a shell command without quitting vi and then restarting it. You can do so in Command mode by preceding the command with :!. Thus, for example, typing :**! pwd** would display the pathname of your current directory, and typing :**! ls** would display the names of all the files in your current directory. After executing a shell command, the editor returns to Command mode.

W25.2.9 Vi, Vim, and Gvim Keyboard Macros

Vi, vim, and gvim offer a variety of *macro* facilities; a macro is a keystroke construction that uses one or more compact set keystrokes to represent another larger number of keystrokes that are substituted for the single or compact set. Macros are used in vi, vim, and gvim for the following reasons:

1. During Insert mode, to construct an abbreviation. For example, in a text file where you use often-repeated blocks of the same text.
2. In Command mode, vi, vim, and gvim commands can be associated or *mapped* to other keys, such as the function keys at the top of the keyboard.
3. Complex commands and their arguments can be triggered by a single keystroke or a shorter sequence of keystrokes.

Here is a brief summary description of the various vi, vim, and gvim macro operations, two of which are covered in the indicated subsections below:

**Text abbreviation** (Section W25.2.9.1), which operates in Insert mode. An abbreviation works only in vi, vim, and gvim Insert mode.

**Keystroke mapping** (Sections W25.2.9.2 and W25.2.9.3), which can operate either in Insert mode or in Command mode, and uses the map! and map Last Line mode commands. Once defined, a map! sequence is triggered only in Insert mode, and a map sequence is triggered only in vi, vim, and gvim Command mode.

**Text-buffer execution**, which operates only in Command mode. Once text has been placed in any of the named text buffers, that text can be executed as if it were a sequence of vi, vim, and gvim commands.

In the following sections, we will describe and give examples of some of the vi, vim, and gvim macro facilities, and also give an additional example of a specialized vim macro feature that can be used in gvim as well. Table W25.8b summarizes the uses of the abbr, map!, and map commands.

*W25.2.9.1 Text Abbreviation Macros Used in Insert Mode*

To save keystrokes while entering text, in Last Line mode, use the abbr(eviate) or just ab command.

It has the following general syntactic form:

**:ab[br] [abbreviation abbreviated]**

where:

**:** gets you into Last Line mode;

**[ ]** designates optional components;

**ab** or **abbr** is the command for creating an abbreviation;

**abbreviation** is a valid string of contiguous (no spaces allowed) characters; and

**abbreviated** is the substitute text you want to be placed in the buffer.

Text abbreviations can be canceled with the Last Line mode unabbr command, followed by typing the abbreviation you want to cancel. Also, if you just type abbr in Last Line mode, you get a listing of all the abbreviations that are active.

To use the abbreviation, when you are in Insert mode, whenever you type the string that represents abbreviation and precede as well as follow it by a nonalphanumeric character, the substitution will take place.

The editor will examine the character before and the next character after you type the abbreviation to see if it’s nonalphanumeric or underscore, and if so, abbreviation will be erased and the string that represents abbreviated will be substituted for it. Also, you are no longer in Insert mode.

For example, in Last Line mode, if you type ab kts Know this stuff! and then press <Enter>, kts is the abbreviation. Then anywhere in Insert mode, when you type sts and precede it and follow it by pressing the space key, the left or right arrow keys (all of which yield nonalphanumeric characters and are not the underscore keys on the keyboard for our system), the string Know this stuff! will be substituted on that line, and you will no longer be in Insert mode.

Note: With abbr, your text appears as you type it, and no substitution is performed on abbreviation until you type nonalphanumeric characters before it and after it. As shown in the next section, this is different from keystroke mapping using the map! or map commands.

The following are some useful abbreviations for Python program file creation:

**:ab 1 #!/usr/local/bin/python**

**:ab 2 from Tkinter import \***

**:ab 3 import os**

**:ab 4 import sys**

*W25.2.9.2 Keystroke-Mapping Macros Used in Insert Mode*

map, shown in the next subsection, works on characters that are typed in Command mode, and map!, shown in this subsection, works on characters that are typed in Insert mode.

As shown in the previous section, abbr won’t substitute text until you type a nonalphanumeric before and after the abbreviation string. Notice the editor echoes each character of the abbreviation as you type it, just in case you really want the string of characters that represents abbreviation to be an actual string of characters that you want in your text. Keystroke mapping works in a more keystroke- and time-dependent way. Keystroke mapping used in Insert mode is handled by the Last Line mode map! command, which takes the following general form:

**:map! [substitution substituted]**

where:

**:** gets you into Last Line mode;

**[ ]** designates optional components;

**map** is the command for creating a keyboard mapping;

**substitution** is a valid string of contiguous (no spaces allowed) characters; and

**substituted** is the substitute text you want to be placed in the buffer.

For example, in Last Line mode, if you type **map! ts This will save you time!** and then press <Enter>, ts is the substitution. Then, anywhere in Insert mode, when you type ts in a short amount of time (under approximately half a second), the string This will save you time! will be substituted on that line and you will still be in Insert mode. If you type more slowly, the literal string ts will be inserted.

The keystroke sequence <Ctrl+V> will let you escape the mapping, as long as you precede the macro with it. So no matter how fast you type in <Ctrl+V> ts, you get the literal string ts inserted.

Remapping abbreviations can be canceled with the Last Line mode unmap! command, followed by typing the substitution you want to cancel. Also, if you just type map! in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

*W25.2.9.3 Keystroke-Remapping Macros Used in Command Mode*

Command-mode remapping is accomplished with the map Last Line mode command.

The general form of the map command is as follows:

**:map [substitution substituted]**

where:

**:** gets you into Last Line mode;

**[ ]** designates optional components;

**map** is the command for creating a keyboard mapping;

**substitution** is a valid string of contiguous(no spaces allowed) characters; and

**substituted** is the substitute text you want to be placed in the buffer.

Some editor command keys cannot be remapped in Command mode. Two examples of these keys are : (colon) and u.

Remapping substitutions can be canceled with the Last Line mode unmap command, followed by typing the remapping you want to cancel. Also, if you just type map in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

As an example, in Last Line mode, if you type **map #8: wq<Ctrl+V><Ctrl+Enter>** and then press **<Enter>**, the function key <F8> at the top of your keyboard is the substitution. The substituted is the command to write the buffer to a file and quit the editor. The <Ctrl+V> and <Ctrl+Enter> keystrokes are the way to enter control characters on the command line, in this case the <Enter> key at the end of the command. After this mapping is done, anytime you are in Command mode, when you press the function key <F8>, the buffer will be written to the default file and you will exit the editor.

Another interesting and useful example is the following map command, which can be placed in your **.exrc** file, so that you can use the function key <F3> during editor sessions to generate a skeleton C program construct:

**map #3 ^[i#include stdio.h ^Mmain(argc, argv) ^M int argc;^M char #argv[];^M{^M}^M^[**

where:

**^[** stands for pressing <Ctrl+V> and then <Esc>; and

**^M** stands for pressing <Ctrl+V> and then the <Enter> key

The relative number of spaces in this map command definition controls the indentation of the skeleton construct. Also, the ^M entries put each of the skeleton construct components on a new line.

*W25.2.9.4 Vim/Gvim Macro Example*

Here is a repeat of Practice Session W25.4, slightly enlarged, that uses a vim-specific macro command sequence to accomplish the same thing that Practice Session W25.4 did, but in another way.

Practice Session W25.6

Step 1: From the shell prompt, type **vim Mintos2** and then press **<Enter>** on the keyboard.

Step 2: In vim, type **A** and then type the following 10 lines of text, each on its own line:

**computer scientists**

**students**

**hackers**

**systems analysts**

**newbies**

**Linux Mint gurus**

**computer programmers**

**systems administrators**

**network administrators**

**Linux users**

Step 3: Press **<Esc>**, then place the cursor anywhere on the first line of text.

Step 4: Type **q a**. This puts you in record mode and associates the macro you are about to record with the **a** key.

Step 5: Type **I**. The cursor is now at the start of the first line in Insert mode.

Step 6: Type **Linux Mint is the operating system of choice for** with a single space after the **r** in the word **for**. Press **<Esc>**.

Step 7: Place the cursor anywhere on the second line of text.

Step 8: Type **q**. This ends record mode.

Step 9: Type 9**@a** This “plays back” the macro defined with the **a** key nine times, once on each of the lines below the first line, inserting the text string Linux Mint is the operating system of choice for.

Step 10: Save the file, print it out, and memorize its contents.

W25.3 THE EMACS EDITOR

The emacs editor is the most complex and customizable of the Linux text editors, and it gives you the most freedom, flexibility, and control over the way you edit text files. It can format text for very specific technical applications, such as program source code development, more effectively than a word processor. Its use in that application makes the process of program development more efficient. In addition, from within the emacs program (deploying multiple windows) you can accomplish a wide variety of personal productivity and operating system tasks, such as sending e-mail and executing shell commands and scripts. But along with more control, specificity, and capabilities comes some a much steeper learning curve, which brings with it a more complex keystroke command structure. This complexity can be offset in part for some users, and totally for others, by using the graphical forms of input and command execution, that we will emphasize in the sections that follow.

As mentioned above, if you cannot run a “graphical” emacs (as we do exclusively by running it in an interactive shell, GUI window in this section) because you are working in a login shell, text-only console or terminal, no worries. You can still gain access to the Menu Bar at the top of the emacs screen by pressing <Esc> on the keyboard and then pressing the single back quote (`) key. You can then descend through the menu bar choices by pressing the letter key of the menu choice you want to make. For example, pressing the f key gives you access to the File pull-down menu choices, and then pressing the s key allows you to save the current buffer. The ways of descending and ascending these menus, and making menu choices, is rather intuitive, and we suggest you experiment as much as is reasonably possible with the non-graphical form of emacs. We present a Problem at the end of this chapter that asks you to do this.

Unfortunately, you cannot access the speed button bar menu choices from within a text-only display of emacs.

To stress how the keyboard keys are used in graphical GNU emacs, we repeat the following note shown at the beginning of this chapter here:

1. Pressing the Escape key is signified as <Esc>

2. Pressing the Enter key is signified as <Enter>

W25. Pressing the <Ctrl> key in combination with another single key is signified as <Ctrl+X>, where you hold down the <Ctrl> key and press the X key (or any valid key for that combination) at the same time.

4. Pressing the <Alt> key in combination with another single key is signified as <Alt+X>,

where you hold down the <Alt> key and press the X key (or any valid key for that combination) at the same time.

5. A variant of W25. and 4. is shown as <Ctrl+X> a [b], where you first press and release <Ctrl> and X simultaneously, and then press the a key, and optionally press the b key (or any valid combination of single keys or strings of characters.

6. In GNU emacs for Linux , the Meta key that is referred to in much of the literature on GNU emacs, is the <Alt> key.

It is important to realize before you begin that there are some common terms used in nano, vi, vim, and gvim (the editors from the previous sections) and emacs, that describe the facilities of each editor. But the terms do not necessarily have the same meaning between the major families of editor.

As you saw in Section W25.1.2, with vi, vim, and gvim, you can’t immediately begin to enter text into the file you are editing. You have to be in Insert mode to do that, that’s what typing A as the second step is doing. Vi, vim, and gvim have modes. In nano and GNU emacs, you can start typing text into the file immediately.

Nano and emacs are *modeless* editors.

Vi, vim, and gvim operate in three distinct modes: Command mode, Insert mode, and Last Line mode. Emacs is a *modeless* editor in the sense that, when you launch emacs, you do not have to switch modes to immediately type characters on the keyboard and enter text into a buffer, or change modes to save the buffer to a file.

For example, emacs does have major modes of operation, such as *Lisp mode*, *Python mode*, and *C mode*, but they are for the special formatting of text and for specialized operations when editing files for use in those language applications. This is different from allowing you to switch between significant forms of action in the editor, as the vi, vim, and gvim Command, Insert, and Last Line modes do. The keystroke command syntax itself in emacs is different and more complex than in vi, involving use of the <Ctrl> and <Alt> prefix characters, as previously noted. The emacs concepts of *point* and the cursor location are also more refined and specific than in vi. In emacs, the point is the location in the buffer where you are currently doing your editing; the point is assumed to be at the left edge of the cursor, or always between characters or white space (what you enter into a text file when you press the space bar). This difference becomes an important issue when you want to use the cut/copy/paste operations. In vi, yanking removes text from the main buffer, much like cutting/copying, whereas in emacs yanking is more like pasting into the main buffer. The concept of a buffer is very important in emacs, and is very much the same in emacs as it is in vi.

Currently, there is one major “brand” of emacs for Linux : GNU emacs. We use the graphical form of GNU emacs version 24.W25.1 in Linux , running in its own frame launched from an interactive shell, in the following illustrations, exercises, practice sessions, and problems.

A complete summary of emacs commands is given in Table W25.9, which we conveniently place at the end of this chapter, so that you can print it out and use it as a handy reference when you are editing with emacs..

W25.3.1 Launching Emacs, Emacs Screen Display, General Emacs Concepts and Features

The general syntax for launching the emacs program from the command line in a console window is as follows (anything enclosed in square brackets [ ] is optional):

**emacs [options][file(s)]**

**Purpose:** Allows you to edit a new or existing file(s)

**Output:** With no options or file(s) specified, emacs runs and begins or opens on the Welcome Screen buffer

**Commonly used options/features:**

**+n** Begin to edit file(s) starting at line number **n**

**-nw** Run emacs without opening a window, useful in an elementary GUI environment

**emacs file1 file2 file3** Open three buffers in emacs on three different files at the same time

For example, if you run the emacs program by typing emacs alien in a terminal window in Linux , emacs launches and shows the Welcome Screen buffer.

A brief description of the major components of the emacs screen display labeled in Figure W25.5 is as follows (note: items J, A, B, D, and C are found on what is called the *mode line*):

A. Name of the current buffer: This is the name of the entity or “file” you are editing in this window. In Figure W25.5, the name of the buffer is **alien**.

B. Major and minor mode: Different major modes are used to edit different kinds of files, like C programs, Lisp, or HTML, and special configurations of the major modes define the minor modes. In Figure W25.5, only the major mode Fundamental is shown, with no minor mode set.

C. Percentage of the text shown on screen: This shows how much of the text in the buffer is seen on screen. In Figure W25.5, all of the text in the current buffer is shown on screen.

D. Current line number: The line location of the cursor in the current buffer is displayed here.

E. Minibuffer: Information and questions/prompts from emacs appear here. In Figure W25.5, New File is displayed because we just opened a new file.

F. Speed button bar: This allows you to do quick, common operations graphically. In Figure W25.5, the File Save icon is pointed out.

G. Menu bar: This gives you pull-down menus that contain all of the important emacs operations.

H. Text: The actual text you are editing appears here.

I. Scroll bar: The scroll bar allows you to graphically scroll or move through the text.

J. Status indicator: Two-character codes are used to tell you about your file. In Figure W25.5, a U- and two hyphens (U---) indicate that the file has not changed in emacs and is the same as the version saved to disk, and that you can work on the file.

In-Chapter Exercises

W25.11. Launch emacs on your Linux system, and identify components A through J of the emacs screen display.

W25.12. If you launch emacs on your Linux system by typing only the word emacs in a terminal or console window (without changing any of the defaults!), how does your screen display differ from what is shown in Figure W25.5? What emacs commands or menu choices can you make to close the emacs Welcome Screen buffer, and begin to edit in a new, completely blank buffer?

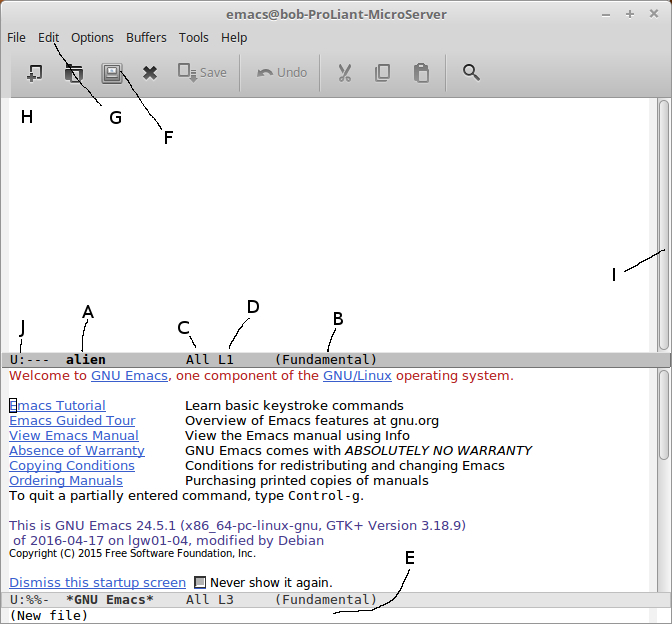


Figure W25.5 First GNU emacs Screen Display

To close the Welcome Screen buffer display that opens in the bottom window of the emacs frame, while the cursor is flashing in the top window, from the top emacs menu, make the pull-down menu choice **File** > **Remove Other Windows**. Once you do this, you will only have one buffer shown in the screen display.

W25.3.1.1 Emacs Help

Emacs provides a wide variety of help commands, all accessible through the key sequence <Ctrl+H> or graphically with the function key <F1>. You can also type <Ctrl+H> <Ctrl+H> to view a list of help commands. You can scroll the list with <Space> and <Del>, then type the help command you want. To cancel, type <Ctrl+G>. Many help commands display their information in a special help buffer. In this buffer, you can type <Space> and <Del> to scroll and press <Enter> to follow hyperlinks.

The following are the most general ways of obtaining help on a topic or command:

**<Ctrl+H>** **a topic(s)** **<Enter>**

This searches for commands whose names match the argument **topic(s)**. The argument can be a keyword, a list of keywords, or a regular expression.

**<Ctrl+H> i d m emacs <Enter> i topic <Enter>**

This searches for **topic** in the indices of the emacs Info manual, displaying the first match found. Press **,** (comma) to see subsequent matches. You can use a regular expression as a topic.

**<Ctrl+H> i d m emacs <Enter> s topic <Enter>**

Similar to <Ctrl+H> … I topic,, but searches the text of the emacs manual rather than the indices.

**<Ctrl+H> <Ctrl+F>**

This displays the emacs FAQ, using Info.

**<Ctrl+H> p**

This displays the available emacs packages based on keywords.

A summary of help command syntax is found in Table W25.10.

|  |  |
| --- | --- |
| <Ctrl+H> a topics <Enter> | Display a list of commands whose names match topics (apropos-command). |
| <Ctrl+H> b | Display all active key bindings—minor mode bindings first, then those of the major mode, then global bindings (describe-bindings). |
| <Ctrl+H> c key | Show the name of the command that the key sequence key is bound to (describe-key-briefly). Here c stands for “character.” For more extensive information on key, use <Ctrl+H> k. |
| <Ctrl+H> d topics <Enter> | Display the commands and variables whose documentation matches topics (apropos-documentation). |
| <Ctrl+H> e | Display the **\*Messages\*** buffer (view-echo-area-messages). |
| <Ctrl+H> f function press <Enter> | Display documentation on the Lisp function named function (describe-function). Since commands are Lisp functions, this works for commands too. |
| <Ctrl+H> h | Display the **HELLO** file, which shows examples of various character sets. |
| <Ctrl+H> i | Run Info, the GNU documentation browser (info). The emacs manual is available in Info. |
| <Ctrl+H> k key | Display the name and documentation of the command that key runs (describe-key). |
| <Ctrl+H> l | Display a description of your last 300 keystrokes (view-lossage). |
| <Ctrl+H> m | Display documentation of the current major mode (describe-mode). |
| <Ctrl+H> n | Display news of recent emacs changes (view-emacs-news). |
| <Ctrl+H> p | Find packages by topic keyword (finder-by-keyword). This lists packages using a package menu buffer. |
| <Ctrl+H> P package <Enter> | Display documentation about the package named package (describe-package). |
| <Ctrl+H> r | Display the emacs manual in Info (info-emacs-manual). |
| <Ctrl+H> s | Display the contents of the current syntax table (describe-syntax). The syntax table says which characters are opening delimiters, which are parts of words, and so on. |
| <Ctrl+H> t | Enter the emacs interactive tutorial (help-with-tutorial). |
| <Ctrl+H> v var <Enter> | Display the documentation of the Lisp variable var (describe-variable). |
| <Ctrl+H> w command <Enter> | Show which keys run the command named command (where-is). |
| <Ctrl+H> C coding <Enter> | Describe the coding system coding (describe-coding-system). |
| <Ctrl+H> C <Enter> | Describe the coding systems currently in use. |
| <Ctrl+H> F command <Enter> | Enter Info and go to the node that documents the emacs command command (Info-goto-emacs-command-node). |
| <Ctrl+H> I method <Enter> | Describe the input method method (describe-input-method). |
| <Ctrl+H> K key | Enter Info and go to the node that documents the key sequence key (Info-goto-emacs-key-command-node). |
| <Ctrl+H> L language-env <Enter> | Display information on the character sets, coding systems, and input methods used in language environment language-env (describe-language-environment). |
| <Ctrl+H> S symbol <Enter> | Display the Info documentation on symbol symbol according to the programming language you are editing (info-lookup-symbol). |
| <Ctrl+H> | Display the help message for a special text area, if the point is in one (displaylocal-help). (These include, for example, links in **\*Help\*** buffers.) |

Table W25.10 Summary of emacs Help Command Syntax

In-Chapter Exercise

13. Use the emacs command **<Ctrl+H> i d m emacs <Enter> s topic <Enter>** to search for the following terms: point, minibuffer, modes, keys

Then write a brief description, in your own words, of each of these terms, based upon what the help provided by emacs yields.

*W25.3.1.2 Graphical Features*

From a beginners perspective, the most useful graphical features of emacs are the menu bar and speed button bar, seen in Figure W25.5 as F and G. These features utilize all of emacs’s functionality with a graphical style of interaction that is most agreeable to a novice, and also to the experienced user as well.

Note that when a menu choice is grayed out, that means it is not available at the current level you are operating at in emacs. Depending on your display, this may be somewhat difficult to determine.

Following is a brief description of what tasks each menu bar item accomplishes-

File: Facilities for opening, saving, and closing buffers, files, windows, and frames

Edit: Means to modify text in buffers

Options: Facilities to make configuration changes

Buffers: A pull-down menu listing of the currently open buffers

Tools: File and application functions

Help: Extensive documentation and on-line manual for emacs

The speed button bar contains single-button presses for (1) file and buffer operations; (2) common text-editing operations, such as cut and paste; and (3) printing, searching, and changing preferences.

In-Chapter Exercise

14. In emacs, repeat as much of Practice Session W25.4 as you can, using only graphical menu choices presented to you on the emacs menu bar or speed button bar. Name the file emacsg. Which editor (vi or emacs) do you prefer when redoing the operations in Practice Session W25.4, and why?

W25.3.1.2.1 Buffers, File, Windows, and Frames

The most important concept in emacs is that of a buffer, or text object that is currently being edited by emacs. This is different from a file, which is a text object stored on disk. The differentiation is made, in simple terms, because (1) the object currently being modified and viewed in emacs is not the same object stored on disk, if you have not yet saved your edits; and (2) emacs can work on text objects that are not files and never will be, such as the output from commands typed on the Linux command line. When you first launch emacs on the command line, and use the option to specify a file to edit, you are looking into the buffer created by emacs for that file, in what is generally known as an emacs frame. That frame may contain only a single window open, that allows you to see the buffer contents. A frame consists of one window, or possibly many windows, with the pull-down and speed button bar menus, the mode line, and a minibuffer.

In-Chapter Exercises

15. When you launch emacs specifying a filename, and close the Welcome screen, how many buffers are open? What are their names? How did you find this out? How do you shift between working in different open buffers? How do you open a new frame on screen? How do you close a frame?

W25.3.1.2.2 Point, Mark, and Region

The second most important concept in emacs is that of the *point* and *mark*, and the *region* of text they demarcate. The point is located in the white space before the character the cursor is highlighting. The mark, set by placing the cursor over a character and then holding down <Ctrl+Space> or <Ctrl+@>, is also in the white space before the character the cursor is highlighting. The region or area of text you want to manipulate in operations such as cutting and copying, is all text between the point and the mark. For example, in the line of text Now is the time for all good men, if the cursor is on or highlighting the N in the word Now (the point is in the white space before the N), and the mark has been set before the character i in the word time by placing the cursor on the letter i and holding down <Ctrl+Space>, then the region is defined as Now is the t.

To exit from emacs without saving any of the buffers, make the pull-down menu choice **File > Quit** or type <Ctrl+X>/<Ctrl+C> on the keyboard.

In-Chapter Exercises

16. In a new emacs file, type in the following text-

Now is the time for all good men

and then place the flashing cursor on the letter N. Define the region by holding down <Ctrl+Space> and then set the mark by using the arrow keys on the keyboard to place the cursor on the n in the word men. What is the region defined as? If you make the speed button bar choice “Copy”, what is copied to the paste buffer? How did you find this out?

17. What signifies the region graphically on a Linux system running with a GUI desktop?

W25.3.2 How to Use Emacs to Do Shell Script File Creation, Editing, and Execution

The following practice session shows how to create a file to define aliases, or command name substitutes, that allow you to type DOS command names at the Linux shell prompt to execute some of the common Linux file maintenance operations. DOS commands are similar to Linux commands but are used in the Windows operating system environment. As you will see in the practice session, you can use an efficient combination of keyboard typing and graphical interaction to work with emacs.

In this section, we assume that you are running the default shell on Linux , the Bourne Again shell, (Bash). Also, it is assumed in these practice sessions, exercises, and problems that you are creating and editing files in your home directory, where you have unaliased all of the aliases.

Practice Session W25.6

Step 1: At the shell prompt, type **emacs alien** and then press **<Enter>**.

The emacs screen appears in your display, similar to Figure W25.5. Close the Welcome Screen as noted in Section W25.3.1, or by making the Welcome screen menu choice “Dismiss this startup screen”.

Step 2: Type **# DOS aliases** and then press **<Enter>**.

Step 3: Type **alias del=rm** and then press **<Enter>**.

Step 4: Type **alias dir=’ls –la’** and then press **<Enter>**.

Step 5: Type **alias type=more** and then press **<Enter>**.

Step 6: Hold down the **<Ctrl+X>**, and then hold down **<Ctrl+S>** to save your file with the name **alien**. The display of your text should appear similar to Figure W25.6.

Step 7: Hold down **<Ctrl+X>**, and then hold down **<Ctrl+C>** to gracefully exit from emacs and return to the Bash shell prompt.

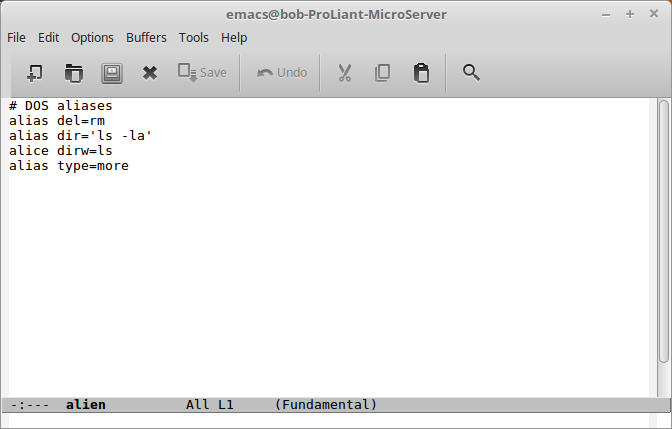


Figure W25.6 File alien After Step 6.

In-Chapter Exercises

18. How did you get the Bash aliases created in the file alien in Practice Session W25.6 to work? List the specific steps you used to do this. The assumption here is that you are working in a GUI desktop system, and you are not in a login shell when you test the aliases.

19. As shown in Practice Session W25.6, use emacs to create a text file that contains the following Bash aliases-

dir='ls -l'

rename='mv'

spr='lpr -Pspr'

lt='ls -ltr'

page='more'

Test the aliases on your system. Note that if your default printer has a designation other than spr, substitute the printer designation on your system. Then, as shown in Chapter 2, Section 6 in the printed book, **unalias** all of the aliases created in this Exercise.

W25.3.3 Visiting Files, Saving Files, and Exiting

Linux stores data permanently in files, so a vast majority of the text a novice user, or even an advanced user, will edit with emacs, comes from a file and is saved in a file. To edit a file while running emacs, you need to read the file into a buffer and prepare that buffer containing a copy of the file’s text. This is called *visiting* the file. (Note: The emacs editing commands work on the text in the buffer inside emacs. Your changes are written to the file itself, by default only when you save the buffer to the file.)

In addition to visiting and saving files, emacs can delete, copy, rename, and append to files, keep multiple versions of them, and operate on file directories.

The following are some of the basic operations you can do to visit files, save them, and then exit gracefully from emacs:

\* Visiting a New File

To visit a new file from within emacs, make the pull-down menu choice **File>Visit New File**. In the Name bar that appears in the Find File window on screen, type in a new file name and then make the choice OK. If the only buffer open is the welcome window, it will close, and you will be editing a buffer named with the new file name.

You can do the same thing by typing <Ctrl+X> <Ctrl+F>. Then in the minibuffer, type in the file name.

\* Saving to a File without Quitting Emacs

After you have entered some text into the current buffer, make the pull-down menu choice **File> Save**.

You can do the same thing by typing <Ctrl+X> <Ctrl+S>.

\* Saving to a File with Unsaved Changes and Quitting Emacs:

If you make unsaved changes to a buffer, and make the pull-down menu choice **File>Quit**, emacs puts a Question dialog box on screen asking you the following:

Save file ? Yes No

View This Buffer

View Changes in This Buffer

Save This but No More

Save All Buffers

No for All

If you make the last menu choice, you are presented with one additional Question Dialog box informing you that Modified buffers exist; exit anyway?

You can do any of these, depending on what you want to accomplish. You get the same choices, although they are less descriptive, when you type <Ctrl+X> <Ctrl+C> for the unsaved buffer.

Experiment with all of the above editing and saving files and buffers methods, to establish a personal operating procedure for yourself when using emacs.

W25.3.4 Cursor Movement and Editing Commands

In addition to general purpose commands, emacs has some important cursor movement and editing commands that allow you to move quickly and easily around the text and make changes. Some of the most important of these commands are listed in Tables W25.12 and W25.13.

|  |  |  |  |
| --- | --- | --- | --- |
| **Entity to Move Over** |  | **Backward** | **Forward** |
| Character |  | <Ctrl+B> | <Ctrl+F> |
| Word |  | <Alt+B> | <Alt+F> |
| Line |  | <Ctrl+P> | <Ctrl+N> |
| Go to line beginning (or end) |  | <Ctrl+A> | <Ctrl+E> |
| Sentence |  | <Alt+A> | <Alt+E> |
| Paragraph |  | <Alt+{> | <Alt+}> |
| Page |  | <Ctrl+X> [ | <Ctrl+X> ] |
| Sexp |  | <Ctrl+Alt+B> | <Ctrl+Alt+F> |
| Function |  | <Ctrl+Alt+A> | <Ctrl+Alt+E> |
| Go to buffer start (or end) |  | <Alt+<> | <Alt+>> |
| Scroll to next screen | <Ctrl+V> |  |  |
| Scroll to previous screen | <Alt+V> |  |  |
| Scroll left | <Ctrl+X> < |  |  |
| Scroll right | <Ctrl+X> > |  |  |
| Scroll current line to center, top, bottom | <Ctrl+L> |  |  |
| Go to line | <Alt+G> g |  |  |
| Back to indentation | <Alt+M> |  |  |

Table W25.12 Entities to Move Over

|  |  |  |  |
| --- | --- | --- | --- |
| **Entity to Kill** |  | **Backward** | **Forward** |
| Character (delete, not kill) |  | <Del> | <Ctrl+D> |
| Word |  | <Alt+Del> | <Alt+D> |
| Line (to end of) |  | <Alt+0><Ctrl+K> | <Ctrl+K> |
| Sentence |  | <Ctrl+X> DEL | <Alt+K> |
| Sexp |  | <Alt+-> <Ctrl+Alt+K> | <Ctrl+Alt+K> |
| Kill region | <Ctrl+W> |  |  |
| Copy region to kill ring | <Alt+W> |  |  |
| Kill through next occurrence of char | <Alt+Z> char |  |  |
| Yank back last thing killed | <Ctrl+Y> |  |  |
| Replace last yank with previous kill | <Alt+Y> |  |  |

Table W25.13 Entities to Kill

Practice Session W25.7 illustrates the use of a mixture of keystroke commands and graphical methods in emacs, and lets you edit the file **alien** that you created in Practice Session W25.6. In particular, Practice Session W25.7 instructs you to insert the file you created in Practice Session W25.6 into a special file you will create in your home directory, named .bash\_aliases, so that, upon subsequent logins, these DOS-aliased commands will be permanently available. This insertion is achieved with an emacs feature known as the *mini-buffer*. Before you begin Practice Session W25.7, do the following:

Preparatory Step 1. By default, there is a .bashrc file in your home directory. In it, you will find the following lines of code that allow you to use a secondary aliases file to contain user-defined aliases:

# Alias definitions.

# You may want to put all your additions into a separate file like

# ~/.bash\_aliases, instead of adding them here directly.

# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash\_aliases ]; then

. ~/.bash\_aliases

fi

Preparatory Step 2. In your home directory, create an empty file named .bash\_aliases using the touch command, or emacs itself.

If you make a mistake anywhere in the following exercise, you can revert to using the graphical form of editing for expediency (using the mouse and pull-down menus, including undo, inside the emacs window.)

Practice Session W25.7

Step 1: At the shell prompt, type **emacs alien** and then press **<Enter>**. Close the Welcome Screen as noted in Section W25.3.1 . The file you created in Practice Session W25.6 is loaded into the buffer, and your screen display should look similar to the one shown in Figure W25.6.

Step 2: Using the arrow keys, position the cursor to the right of the **‘** character at the end of the third line.

Step 3: Press **<Enter>**

Step 4: Type **alice dirw=ls**

Step 5: Hold down **<Ctrl+A>**  The cursor moves to the beginning of the line.

Step 6: Hold down **<Alt+D>** The word alice has been cut from the buffer.

Step 7: Type **alias**

Step 8: Hold down **<Alt+B>** The cursor moves to the beginning of the word alias.

Step 9: Position the cursor with the arrow keys on the keyboard at the beginning of the first blank line, below the line that reads **alias type=more**

Step 10: Hold down **<Ctrl+Y>** The cut word alice has been put back into the buffer at the start of the line.

Step 11: Use the arrow keys to position the cursor in the space to the right of the word alice if it is not there already.

Step 12: Use the **<Delete>** or **<Backspace>** keys to delete the letters c and e from the word alice.

Step 13: Type **as copy=cp**

Step 14: Hold down **<Ctrl+X>** **<Ctrl+W>**

Step 15: At the Write file: prompt, erase anything on the line with the **<Backspace>** key, and type **alien2** and then press **<Enter>**. Your screen display should now look similar to the one shown in Figure W25.7.

Step 16: Hold down **<Ctrl+H>** and then press the **a** key. The minibuffer area shows a prompt for you to obtain help. Hold down **<Ctrl+G>**. Doing so cancels your help request.

Step 17: Hold down **<Ctrl+X>** **<Ctrl+C>** to quit emacs and return to the shell prompt.

Step 18: From the shell prompt, type **emacs .bash\_aliases** and then press **<Enter>**. The blank contents of your **.bash\_aliases** file now appears in the editing buffer.

Step 19: Position the cursor with the arrow keys on the keyboard on the first blank line in the file. Hold down **<Ctrl+X>** and then press the **i** key on the keyboard. This will allow you to insert the contents of a file into the current buffer at the position of the cursor.

Step 20: In the minibuffer, type **alien2**. The lines of text from **alien2**’s DOS aliases should now be inserted into the file **.bash\_aliases** after and below where you positioned the cursor in Step 19.

Step 21: From the pull-down menu File, make the choice **File>Save (current buffer)**, or use **<Ctrl+X>** **<Ctrl+S>**.

Step 22: Hold down **<Ctrl+X>** **<Ctrl+C>** to quit emacs and return to the shell prompt.

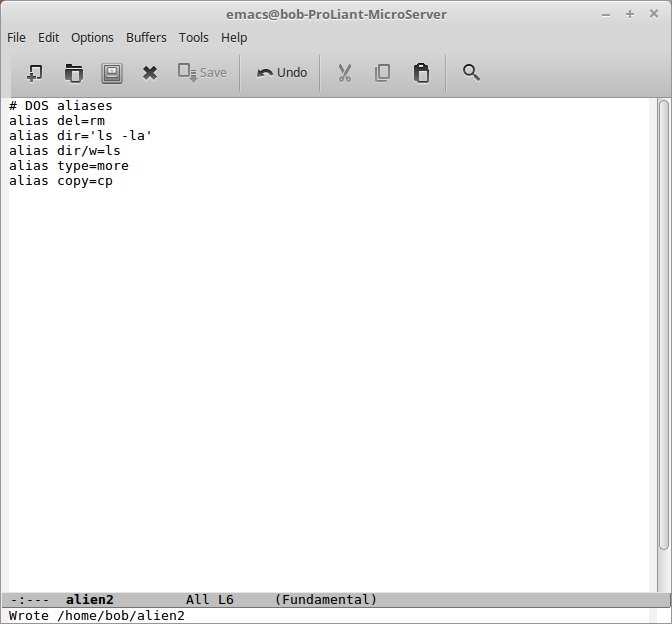


Figure W25.7 Display after Step 15.

In-Chapter Exercises

20. Test the new **.bash\_aliases** file created in Practice Session W25.7 . How did you do this?

The assumption here is that you are working not working in a login shell! In a terminal window, at the shell prompt, type one of the aliased commands, with its appropriate arguments if necessary, and note the results. Which aliases work, and which ones don’t? Why?

21. Use emacs to correct the Bash aliases that do not work in the **.bash\_aliases** file. Then test them. Finally, unalias them.

W25.3.5 Keystroke Macros

The emacs text editor contains a simple-to-use facility that allows you to define keystroke macros, or collections of keystrokes that can be recorded and then played back at any time. This capability allows you to define repetitive multiple keystroke operations as a single command and then execute that command—as many times as you want. The keystrokes can include emacs commands and a series of keystrokes. A macro can also be saved with a name, or even be saved to a file, for use during subsequent emacs editing sessions. Table W25.13 shows a list of some of the most important keyboard macro commands.

For a more complete description and detailed explanation of how to record, edit, list, and delete keystroke macros, see Section W25.3.1.3.5.

Practice Session W25.9 lets you create a new text file using some of the commands presented in Table W25.13.

W25.3.5.1 Practice Session W25.9

Step 1: At the shell prompt, type **emacs datafile** and then press **<Enter>**.

The emacs screen appears on your display.

Step 2: Hold down **<Ctrl+X>** **<Shift+9>**. These actions begin your keyboard macro definition. If you make a mistake anywhere in subsequent steps, simply hold down **<Ctrl+G>** to cancel the current macro definition.

Step 3: Type **1 2 3 4 5 6 7 8 9 10** and then press **<Enter>**.

Step 4: Hold down **<Ctrl+X> <Shift+0>**. These actions end your macro definition.

Step 5: Hold down **<Ctrl+X> e**. Doing so replays the macro that you just defined, placing another line of the numbers 1 through 10 in the buffer.

Step 6: Repeat Step 5 eight more times so that your display looks similar to that shown in Figure W25.8.

Step 7: Hold down **<Ctrl+X> <Ctrl+S>**. These actions save the buffer to the file datafile.

Step 8: Hold down **<Ctrl+X> <Ctrl+C>** to exit from emacs.

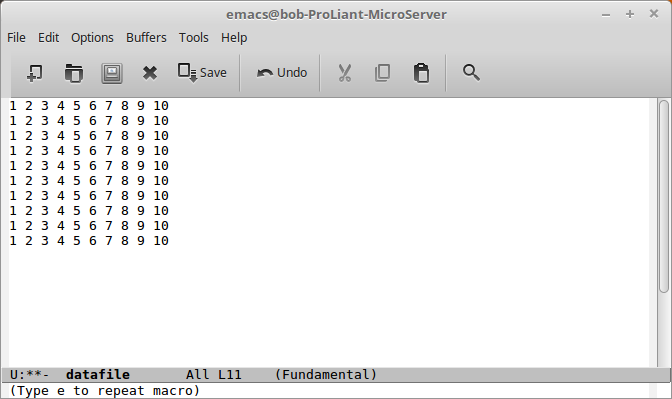


Figure W25.8 Display After Step 6.

W25.3.6 Cut or Copy and Paste and Search and Replace

As we mentioned previously, every word processor has the capability to cut or copy text and then paste that text back into the document and to search for old text and replace it with new text. Because emacs operations can be totally text activated, whereby you use sequences of keystrokes to execute commands, cutting or copying and pasting are fairly complex operations. They are accomplished with the *Kill Ring*, whereby text is held in a buffer by killing it and is then restored to the document at the desired position by yanking it. Global search and replace are somewhat less complex and are accomplished by either an unconditional replacement or an interactive replacement.

The mark is simply a place holder in the buffer. For example, to cut three words from a document and then paste them back at another position, move the point before the first word you want to cut and press <Esc+D> three times. The three words are then cut to the Kill Ring. Because the Kill Ring is a FIFO buffer, you can now move the point to where you want to restore the three words and press <Ctrl+Y>. The three words are yanked into the document in the same order, left to right, that they were cut from the document.

To copy three words of text and then paste them back at another position, set the mark by positioning the point after the three words, and then press <Ctrl+@> at that position. Then reposition the point before the three words; you have now defined a region between the point and the mark. There is only one mark in the document. Press <Esc+W> to send the text between the point and the mark to the Kill Ring; the text is sent, but it is not blanked from the screen display. To restore the three words at another position, move the point there and press <Ctrl+Y>. The three words are restored at the new position. Table W25.11 gives the important kill and yank commands for emacs.

Global search and replace can be either unconditional, where every occurrence of old text you want to replace with new text is replaced without prompting, or it can be interactive, where you are prompted by emacs before each occurrence of old text is replaced with new text. Also, the grammar of replacement can include regular expressions, which we do not cover here.

For example, to replace the word men unconditionally with the word women from the current position of the point to the end of the document, press <Esc+X>, type replace-string, and then press <Enter>. You are then prompted for the old string. Type men and then press <Enter>. You are then prompted for the new string. Type women and then press <Enter>. All occurrences are replaced with no further prompts.

To accomplish an interactive replacement, simply press <Esc+X>, type query-replace, and then press <Enter>. You can then input old and new strings, but you are given an opportunity at each occurrence of the old string to replace it or not to replace it with the new string. Table W25.12 shows the actions that you can take while in the midst of an interactive search and replace.

|  |  |
| --- | --- |
| **Search and Replace Action** | **Keystrokes** |
| Search forward | <Ctrl+S> |
| Search backward | <Ctrl+R> |
| Regular expression search | <Ctrl+Alt+S> |
| Reverse regular expression search | <Ctrl+Alt+R> |
| Select previous search string | <Alt+P> |
| Select next later search string | <Alt+N> |
| Exit incremental search | <Enter> |
| Undo effect of last character | <Del> |
| Abort current search | <Ctrl+G> |
| Interactively replace a text string | <Alt+%> |
| Using regular expressions | <Alt+X> query-replace-regexp |
| Replace this one, go on to next | <Space> or y |
| Replace this one, don’t move | , |
| Skip to next without replacing | <Del> or n |
| Replace all remaining matches | ! |
| Back up to the previous match | ^ |
| Exit query-replace | <Enter> |
| Enter recursive edit (<Ctrl+Alt+C> to exit) | <Ctrl+R> |

Table W25.12 Interactive Search and Replace

Practice Session W25.10 contains further examples of copying and pasting and global search and replace, both unconditional and interactive. Your objective will be to type in one line of text, copy it into the Kill Ring, and then paste it into the document seven times. Then modify the contents of the original line and each pasted line by using both interactive search and replace and unconditional search and replace. Upon completion of Practice Session W25.10, your screen display should look similar to Figure W25.9.

Practice Session W25.10

Step 1: At the shell prompt, type **emacs osfile** and then press **<Enter>**.

Step 2: Type **Windows is the operating system of choice for everyone.**

Step 3: Press **<Ctrl+@>**. The mark is now set at the end of the line you typed in Step 2. Highlight the whole first line with the graphics cursor and left mouse button. This will define the region that will be put in the Kill Ring.

Step 4: Press **<Esc+W>**. This action copies the region to the Kill Ring.

Step 5: Press **<Enter>** to start a new line in the buffer, which should be blank. The cursor should be positioned at the start of this new line.

Step 6: Press **<Ctrl+Y>**. The first line of text is now pasted into the next blank line from the Kill Ring.

Step 7: Repeat Steps 5 and 6 six more times so that you now have eight lines of text in the buffer, all containing the text Windows is the operating system of choice for everyone.

Step 8: Position the cursor on the W in Windows on the first line of the buffer.

Step 9: Save the buffer at this point with **<Ctrl+X> <Ctrl+S>**.

Step 10: Press **<Esc+X>**. Then type **query-replace** and press **<Enter>**. These actions begin an interactive search and replace. The prompt Query replace: appears.

Step 11: Type **everyone** and then press **<Enter>**. The prompt with: appears.

Step 12: Type **students** and then press **<Enter>**. The prompt Query replacing everyone with students: (? for help) appears.

Step 13: Pressing **<Space>** on the keyboard replaces the word everyone on the first line with the word students, and the prompt Query replacing everyone with students: (? for help) appears again.

Step 14: Press **<Enter>**. The prompt Replaced 1 occurrence appears.

Step 15: Position the cursor over the e in the word everyone on the second line of the buffer.

Step 16: Repeat Steps 10–14, interactively replacing the word everyone each time it appears with the words computer scientists, engineers, system administrators, web servers, scientists, networking, and mathematicians on lines 2–8 of the buffer.

Be sure that the second through eighth times you do Step 16, you always position the cursor on the previous line to the current line you want to replace text on!

Step 17: Position the cursor on the W in Windows on the first line of the buffer.

Step 18: Press **<Esc+X>**. Then type **replace-string** and press **<Enter>**. These actions begin an unconditional search and replace. The prompt replace string: appears.

Step 19: Type **Windows** and then press **<Enter>**. The prompt Replace string Windows with: appears.

Step 20: Type **Linux Mint** and then press **<Enter>**. The prompt Replaced 8 occurrences appears. Correct?

Step 21: Save the buffer with **<Ctrl+X><Ctrl+S>**, print it using the facilities available on your computer system, and exit emacs with **<Ctrl+X> <Ctrl+C>**. Your screen display should appear like Figure W25.9.

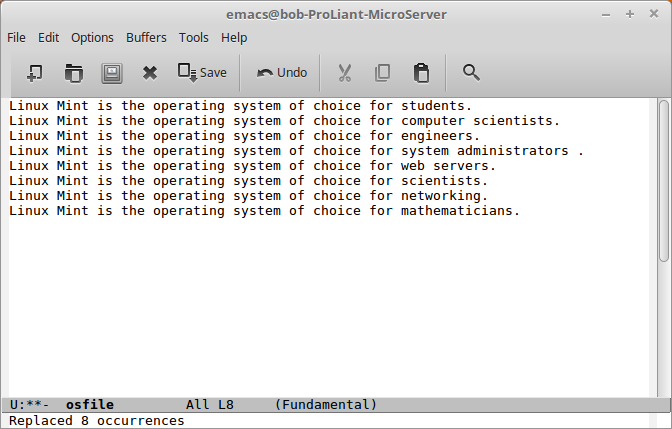


Figure W25.9 Display After Step 21.

The following in-chapter exercises ask you to apply some of the operations you learned about in the previous practice sessions:

In-Chapter Exercises

22. Run emacs and define keyboard macro commands that automatically delete

a. every other word in a line of unspecified length,

b. every other line in a file of unspecified length,

c. every other word and every other line in a file of unspecified length with lines of unspecified length.

23. Write a keyboard macro, as shown in Section W25.3.5, to do everything shown in Steps 10–14 of Practice Session W25.10.

W25.3.7 How to Do Purely Graphical Editing with GNU Emacs

Up to this point in our work, it was possible to use emacs in a single, text-based terminal window, and obtain the results shown. If you connect to Linux by using one of the methods described in Chapter 2 in the printed book, you are likely interfacing with the operating system via GUI desktop system. This would allow you to do all of your emacs work in a graphical environment. For the purposes of learning emacs (if you are using Linux implemented via a GUI on a desktop) you may be able to run emacs in its own frame on your screen display or possibly in several frames on your screen display simultaneously.

W25.3.8 Editing Data Files

The following practice session demonstrates the use of graphical GNU emacs to do some further editing of the datafile created in Practice Session W25.9. The look and feel of GNU emacs, running under GUI desktop default windowing environment in Linux, is very similar to a word processor or desktop publishing application running under any other operating system that has a GUI, such as Windows 10 or Mac OS X. In the practice sessions that follow, we are using GNU emacs 24.5.1.

Practice Session W25.11

Step 1: In a terminal or console window, at the shell prompt, type **emacs datafile** and then press **<Enter>**. Your screen display should look similar to the one shown in Figure W25.8.

Step 2: Use the mouse to position the cursor over the character 1 at the beginning of the tenth line in the buffer, and then click the left mouse button. The cursor is now positioned over the character 1.

Step 3: Click and hold down the left mouse button over the character 1, and then drag the mouse so that the entire tenth line is highlighted, including one character to the right of the 0 in the number 10 at the end of the line. Release the left mouse button.

The whole first line should be highlighted.

Step 4: Position the cursor with the mouse so that the arrow points to the menu choice **Edit** at the top of the emacs screen. Click the left mouse button. A set of pull-down menu choices appears, similar to that shown in Figure W25.10.

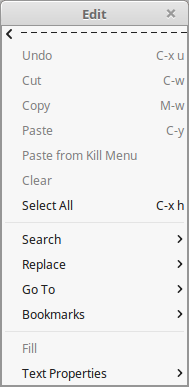


Figure W25.10 Edit Pull-Down Menu

Step 5: Make the **Copy** menu choice. The text that you highlighted (selected) in Step 3 is now held in a temporary buffer.

Step 6: Press **<Enter>**. This opens an eleventh line at the bottom of the buffer.

Step 7: Move the mouse so that the cursor is over the first character position on the eleventh line, and click the left mouse button. The cursor is now in that position in the buffer.

Step 8: Make the pull-down menu choice **Edit>Paste**. You have now pasted the 10 characters from the tenth line in the buffer into the eleventh line in the buffer. Your screen display should now look similar to Figure W25.11.

Step 9: Make the pull-down menu choice **File>Save (current buffer)**. In the Write file: dialog box that appears, save the file in your home directory as **datafile11**, and then make the pull-down menu choice **File>Exit Emacs**.

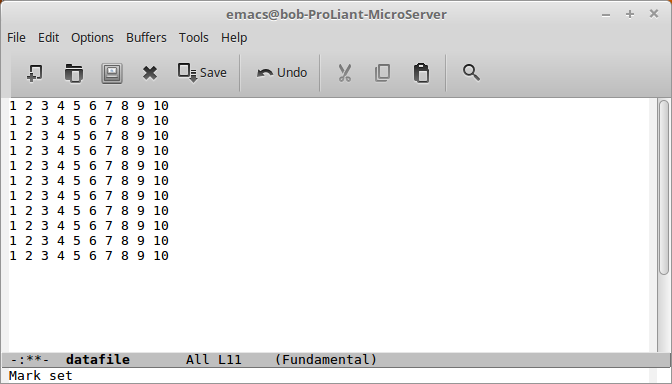


Figure W25.11 Datafile After Editing and Adding an 11th Line

W25.3.9 How to Start, Save a File, and Exit in Graphical Emacs

As illustrated in Practice Session W25.11, GNU emacs can give you a high degree of mouse/GUI command expediency. This method of working on a text file is most efficient for beginners as well as experienced users. In doing Practice Session W25.12, which starts by editing the file you created in Practice Session W25.7, you will be able to compare the speed and ease of operations using keystroke commands to those of mouse/GUI command interaction. Note that, on the pull-down menu shown in Figure W25.10, keystroke commands also are shown for some of the menu choices. Clicking the menu choice button or pressing the keyboard keys would accomplish the same thing. This flexibility adds to the ease of your use of emacs.

Of course, you still have to use the keyboard to enter text!

Practice Session W25.12 lets you edit the file **alien** that you created at the start of this section in Practice Session W25.7. That practice session allowed you to use emacs to create a simple Bash shell script file of shell command aliases. You will now modify it so that it can be used to create aliases for the C shell. You will also modify the existing file **.cshrc** in your home directory so that when you log in and are using the C shell, you have the aliased shell commands for the C shell in the file **alien3** available to you. Before you begin Practice Session W25.12, take the following preparatory steps (modify the **.cshrc** file in your home directory and proceed through this practice session for the C shell rather than the Bash shell):

As with all of the previous Practice Sessions, we assume that you are working in an interactive shell terminal or console window. Also, if the C shell is not installed on your Linux system, install it using your package manager.

Preparatory Step 1: Use the **ls -la** command to find out if you have a **.cshrc** file in your home directory. If you have no **.cshrc** file in your home directory, then use emacs to create a new file named **.cshrc** with no text in it. Then exit emacs, and type **chmod u+x .cshrc** and press **<Enter>**.

Preparatory Step 2: We assume by default that, in your Linux system, Bash is your default shell. To find out which shell you are currently using, type **echo $SHELL** and pressing **<Enter>**. If you are using the C shell, the system will respond with /bin/csh. If you are using the Bash shell, the system will respond /bin/bash.

Preparatory Step 3: If you are not using the C shell as determined in Step 2, switch to the C shell at the existing shell prompt by typing **csh** and then pressing **<Enter>**.

Practice Session W25.12

Step 1: At the shell prompt, type **emacs alien** and then press **<Enter>**. The file that you created in Practice Exercise W25.7 is loaded into the buffer, and the contents of the emacs buffer looks like the one shown in Figure W25.6. Use the cursor and mouse for cursor positioning, and the keyboard keys for text entry and deleting to modify the file so that it looks like this (the proper format of aliases for the C shell):

**#DOS aliases for the C shell**

**alias del rm**

**alias dir ‘ls -la’**

**alias type more**

Step 2: Position the cursor, using the mouse and left mouse button, to the right of the single-quote character (**‘**) at the end of the third line.

Step 3: Press **<Enter>** to open a blank line and put the cursor at the beginning of the line.

Step 4: Type **alice dirw ls**

Step 5: Position the cursor, using the mouse and left mouse button, at character a in alice.

Step 6: Hold down the left mouse button and move the mouse so that the word alice and the following space are highlighted. At the top of the screen, make the **Edit** pull-down menu choice **Cut** to cut the word alice from the buffer.

Step 7: Type **alias**.

Step 8: Move the mouse so that the cursor is over the second a character in the word alias. Click the left mouse button.

Step 9: Press the **<Down>** arrow key on the keyboard twice. The cursor should now be at the beginning of the blank line below the line that reads **alias type more**

Step 10: From the **Edit** pull-down menu, choose **Paste**. The cut word alice has been put back into the buffer at the start of the line.

Step 11: Use the mouse and left mouse button to position the cursor at the end of the word alice, after the character e.

Step 12: Use the **<Delete>** or **<Backspace>** keys to delete the letters c and e from the word alice.

Step 13: Type **as copy cp**

Step 14: Continue moving the cursor to the proper positions and add the necessary characters.

Step 15: From the pull down menu File, choose **Save Buffer As…**

Step 16: In the Write file: dialog box that opens on screen, save the file as **alienW25.**

Step 17: From the File pull-down menu, make the choice **File>Open File**. In the Find file: dialog box that opens, put a check mark in the box that is for Show Hidden Files.

Locate and select the **.cshrc** file you created as an empty (no text in it) file in Preparatory Step 1.(which should be in your home directory), and make the **Open** choice. A new buffer opens on screen containing the blank contents of the **.cshrc** file. Position the cursor anywhere on a blank line in the buffer for the file **.cshrc**.

Step 18: From the File pull-down menu, make the choice **File>Insert file…** In the dialog box that opens, choose **alien3** and insert it. The lines of text from **alien3**’s DOS aliases should now be inserted into the file **.cshrc** at the location you designated in Step 17.

Step 19: From the pull-down menu File, make the choice **File>Save (current buffer)**.

Step 20: Make the pull-down menu choice **File>Exit Emacs** to quit emacs and return to the shell prompt.

Step 21: To test your new **.cshrc** file , do the following. First exit the C shell by typing **exit** and pressing <Enter>. Close and then reopen the terminal window to re-initiate the interactive Bash shell, run the C shell again, and test the new aliases and note the results. For example, if you type **dir**, you should get the results of the **ls -la** command that is executed in the current working directory.

W25.3.10 Emacs Graphical Menus

Figures W25.12 and W25.13 show the contents of another two of the most important pull-down menus in a graphical emacs: Files and Tools. To the right of each pull-down choice is the keystroke command equivalent, if there is one.

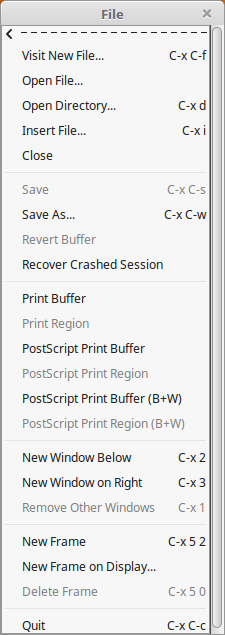


Figure W25.12 Files Pull-Down Menu

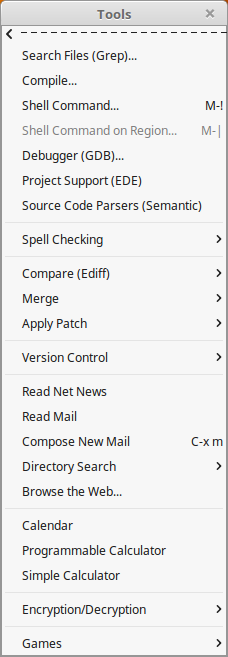


Figure W25.13 Tools Pull-Down Menu

W25.3.11 Creating and Editing C Programs

Besides being a powerful text editor/word processor, emacs can do multiple chores that are useful to a computer user from within the emacs program itself, such as composing e-mail, executing shell scripts, Internet work, and program development in C, HTML, and Java. Since the text for anything more than a trivial program must be generated in a text editor of some sort, it stands to reason that this editor should also be able to compile, link, debug, and keep a record of source code revisions, as well as execute the program itself. This is easily done in emacs using some of its built-in capabilities. These kinds of all-in-one capabilities are present because in the days of character-only terminals and consoles, instead of leaving the editor to accomplish a chore outside of it, you could accomplish common tasks from within the editor. In modern Linux, we can now simply switch between windows and never leave the editor. But it is still very useful to be able to harness some of the multiple capabilities of the program, mainly for the sake of efficiency.

Practice Session W25.13 allows you to type in the source code of a C program and use the special facilities of the editor to properly indent the text, compile and link the source code, and implement revisions according to compile-time errors. You can then execute the program in a terminal window to test it. The purpose of the program is to allow the user to type in an integer, and then another integer, and the first integer will be raised to the power indicated by the second integer.

A note about paths: If the path for the shell you are executing includes the working directory where the compiled and linked executable program is saved by emacs, then you can run the program. Otherwise, you will have to include the path to this directory. For example, emacs saved the source code and executable files in the current working directory /root on our system when we did Practice Session W25.13. Before running the program, since we knew we were running under the C shell, we checked the path variable by typing echo $PATH. The path display included the current working directory where emacs was saving our files. See Chapter 2, Sections 2.7 and 2.8, for information on how to view the path and set the path variable. Also see Chapter 14 for more about Linux tools for the software development process. The source code for the program is as follows:

#include <math.h>

main()

{

float x,y;

printf("This program takes x and y values from stdin and displaysx^y.\n");

printf("Enter integer x: ");

scanf("%f", &x);

printf("Enter integer y: ");

scanf("%f", &y);

printf("x^y is: %6.3f\n", pow((double)x,(double)y));

}

Practice Session W25.13

Step 1: At the shell prompt, type **emacs power.c**. Notice that the major mode for this new buffer is set to C/l mode.

Step 2: Type in the program source code exactly as shown. Use the **<Tab>** key to produce the indentation shown in the C source code. Your emacs screen display should look similar to Figure W25.14.

Step 3: From the pull-down menus, make the choice **File>Save (current buffer)**.

Step 4: From the pull-down menus, make the choice **Tools>Compile…** In the minibuffer, the prompt Compile command: make -k appears. Use the backspace key to erase make -k, and then, to replace it, type g**cc power.c -lm -o power**. A new buffer window appears in the emacs frame, showing the progress of the compilation/linking process.

Step 5: From the pull-down menus, make the choice **Tools>Compile…** In the minibuffer, the prompt Compile command: gcc power.c -lm -o power should appear. Press **<Enter>** to accept this compile/link command.

If you made mistakes in typing the C code, repeat Steps 2 through 5 until you get no error messages that prevent compilation and linkage! The bottom buffer window of Figure W25.14 shows *warning messages*, but not exceptions that prevented compilation and linkage.

Step 6: If all syntax errors have been removed from the power.c source code, you should get a screen display similar to Figure W25.14, which indicates in the bottom buffer window that you have successfully compiled and linked power.c.

Step 7: You can now exit emacs, and in a terminal window test the program by typing **power** on the command line. Remember that the path must be set for the current shell so that executable programs in the directory the file power is in will run.

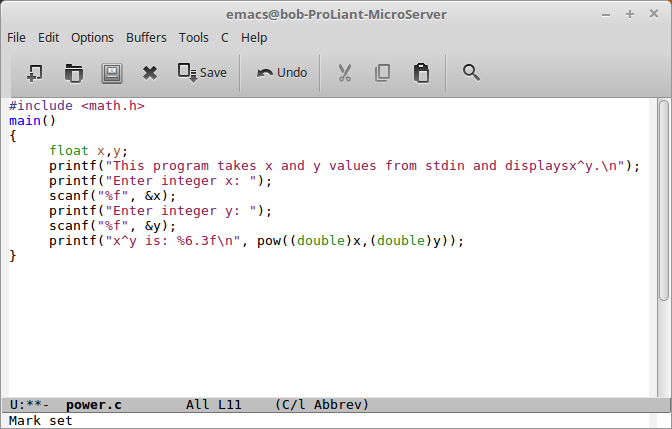


Figure W25.14 Display After Step 2.

W25.3.12 Working in Multiple Buffers

As seen in previous exercises, it is possible to insert one buffer into another and to open windows into different buffers, some of which may not even contain text you want to edit, at the same time. This capability is important when you want to compose the contents of a buffer or file from perhaps many other buffers or files that you have previously created. The following practice session shows you how to create, move between, and copy and paste between several buffers open within one emacs frame.

Practice Session W25.14

Step 1: Create a subdirectory under your home directory named **multi**, and make that subdirectory the current working directory.

Step 2: At the shell prompt, type **emacs newfile**. You should now be editing the buffer **newfile** with a single window.

Step 3: In emacs, make the pull-down menu choice **File>New Window Below**. The frame should now be split horizontally, so that you have two windows, one above the other, both showing the contents of **newfile**.

Step 4: Click with the mouse in the upper window, and then press **<Ctrl+X> 3.** The upper window from Step 3 should now be split vertically into two windows, showing you a total of three windows into the buffer **newfile**.

Step 5: Repeat Step 4 in the lower window of the frame. You should now have four windows showing the contents of the buffer **newfile**. Your screen display should look similar to Figure W25.15. If you did Steps 1–4 incorrectly, you can always use the **File>Remove Other Windows** pull-down menu choice to return you to a single window display, and then try again.

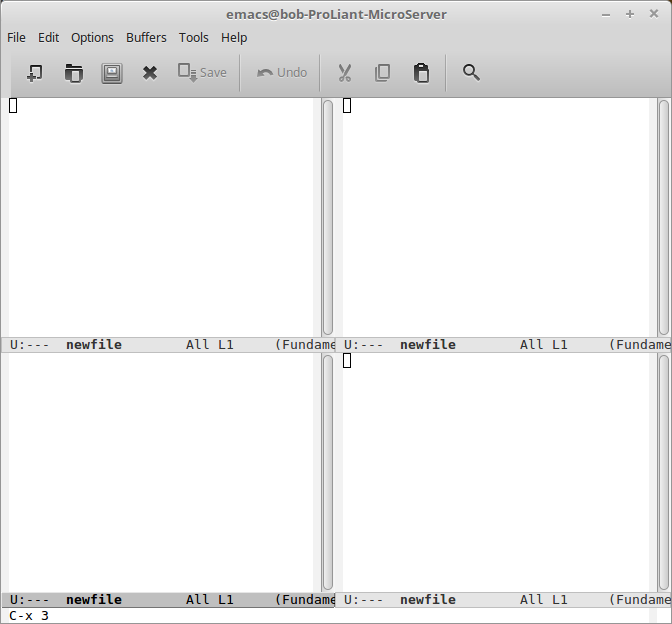


Figure W25.15 Display After Step 5.

Step 6: Click the mouse in the upper-left window and type **1 2 3 4 5**. Then make the pull-down menu choice **File>Save As**. In the Write file: dialog box that appears on screen, type **firstrow** in the Name: box, and then use the Name dialog pane and double left click on the folder **multi**. Then, make the choice **OK**. A new file named **firstrow** is created on disk in the directory named **multi**, and you are still seeing four windows into that buffer.

Step 7: Click the mouse in the upper-right window, position the cursor at the right after the 5, and use the **<Backspace>** or **<Delete>** keys to erase the numbers 1, 2, 3, 4, and 5. Then type **6 7 8 9 10**. Then make the pull-down menu choice **File>Save As**. In the Write file: dialog box that appears on screen, type **secondrow** in the Name: box. The file will be saved in the folder **multi**. Then make the choice **OK**.

Step 8: Click the mouse in the upper-left window. Make the pull-down menu choice **File>Open File**. In the Find file: dialog box that appears on screen, highlight the file **firstrow** in the Name dialog pane. Then make the choice **Open**. You now should have a screen display similar to Figure W25.16, with the upper-left window showing the contents of **firstrow**, and the remaining three windows showing the contents of **secondrow**.

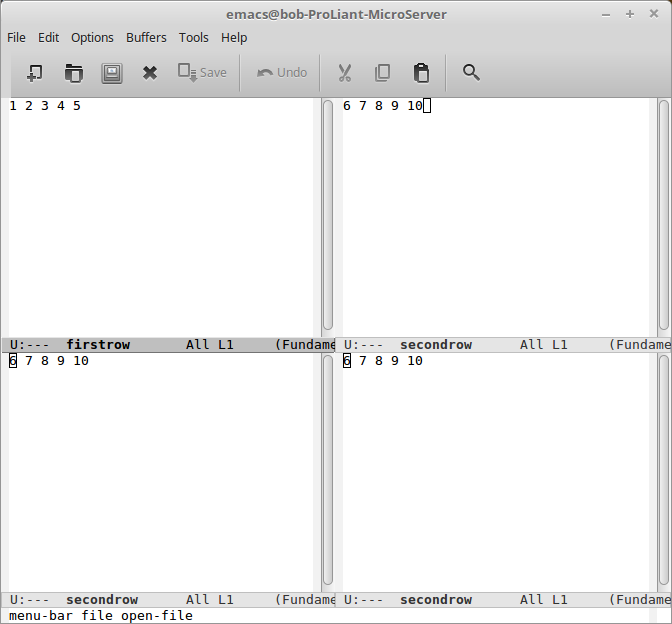


Figure W25.16 Display After Step 8.

Step 9: Click the mouse in the lower-left window, position the cursor to the right of the 0, erase 6, 7, 8, 9, and 10, and type **11 12 13 14 15**. Then make the pull-down menu choice **File>Save As**. In the Write file: dialog box that appears on screen, type **thirdrow** in the Name: box. Then make the choice **OK**.

Step 10: Click in the upper-right window and make the pull-down menu choice **File>Open File**. In the Find file: dialog box that appears on screen, highlight the file **secondrow** in the Name dialog pane. Then, make the choice **Open**.

Step 11: Click in the lower-left window, and make the pull-down menu choice **File>Open File**. In the minibuffer, type **thirdrow**. Your screen display should now look similar to Figure W25.17.

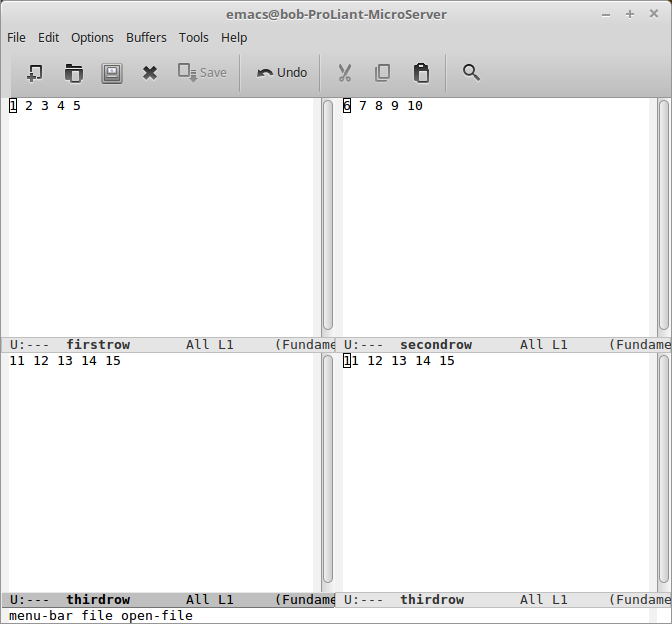


Figure W25.17 Display After Step 11.

Step 12: Click the mouse in the lower-right window, and make the pull-down menu choice **File>Save As**. In the Write file: dialog box, type **four** in the Name: box. Make the choice **OK**. Click in the lower-left window, and make the pull-down menu choice **File>Open File**. In the Find file: dialog box that opens, highlight **thirdrow**, and open it.

Step 13: Click the mouse in the lower-right window, and use the backspace key to erase 11, 12, 13, 14, and 15. Then use the pull-down menu choices **Edit>Copy** and **Edit>Paste** to copy 1 2 3 4 5, 6 7 8 9 10, and 11 12 13 14 15 onto the first three rows of the lower-right window. Your screen display should look similar to Figure W25.18.

Step 14: Finally, with the lower-right window the current window, make the pull-down menu choice **File>Save As**. In the Write file: dialog box, in the Name box, type **four**. Overwrite the old buffer **four**. Then quit emacs without saving any of the buffers.

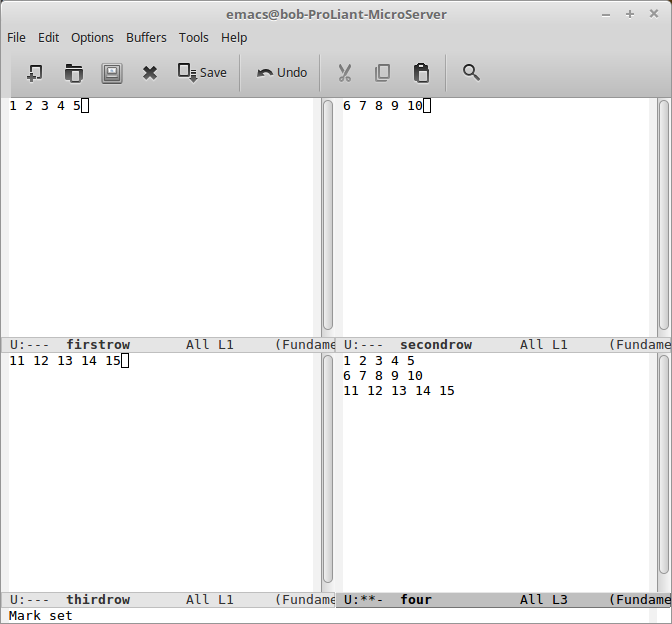


Figure W25.18 Display after Step 13

W25.3.13 Changing Emacs Behavior

This section describes the basic methods of customizing and modifying the behavior of GNU emacs. This includes the following operations:

1. Using the Options menu to modify options.
2. Using Custom (a GUI-based interface) to change preferences and options, and in conjunction with that interface, also using the traditional typed <Alt+X> customize command set.
3. Writing keystroke abbreviations with abbrev.
4. Writing keystroke macro commands, as shown in Section W25.3.5.
5. Redefining keyboard keys.
6. Writing emacs Lisp (elisp) code to customize the behavior of emacs, and entering that code directly into your **~/.emacs** startup configuration file.

All of these operations can make changes to your **~/.emacs** startup configuration file to give you a more customized and personalized emacs session, one tailored to your particular needs and methods of entering text for a particular application.

Be aware that by default, an ~/.emacs configuration file on Linux Mint does not initially exist. But once you begin the procedures for customizing emacs in the sections that follow, that file will be created by emacs.

Also, as will be seen, elisp code is generated by what operations you do. But you don’t really need to know any of the details of how to program in elisp to actually achieve all of the these operations!

The following subsections describe and give examples of all of the given operations. In addition, Tables W25.13 and W25.14 give a summary of the important keystrokes that implement <Alt+X> customize, keystroke abbreviations with abbrev, and writing keystroke macros.

|  |  |
| --- | --- |
| **Customization Action** | **Keystrokes** |
| **Abbrevs** |  |
| Add global abbrev | <Ctrl+X> a g |
| Add mode-local abbrev | <Ctrl+X> a l |
| Add global expansion for this abbrev | <Ctrl+X> a i g |
| Add mode-local expansion for this abbrev | <Ctrl+X> a i l |
| Explicitly expand abbrev | <Ctrl+X> a e |
| Expand previous word dynamically | <Alt+/> |
| **Macros** | |
| Start defining a keyboard macro | <Ctrl+X> ( or <F3> |
| End keyboard macro definition | <Ctrl+X> ) or <F4> |
| Execute last-defined keyboard macro | <Ctrl+X> e or <F4> |
| Append to last keyboard macro | <Ctrl+U> <Ctrl+X> ( |
| Name last keyboard macro | <Alt+X> name-last-kbd-macro |
| Insert Lisp definition in buffer | <Alt+X> insert-kbd-macro |
| Customize variables and faces | <Alt+X> customize |
| **Simple customization with <Alt+X> customize** | |
| (global-set-key (kbd "<Ctrl+C> g") 'search-forward) |  |
| (global-set-key (kbd "<Alt+#>") 'query-replace-regexp) |  |

Table W25.13 Ways to Change emacs Behavior

|  |  |  |
| --- | --- | --- |
| **Keystrokes** | **Command Name** | **Action** |
| <Ctrl+X> ( | kmacro-startmacro | Start macro definition. |
| <F3> | kmacro-startmacro-or-insertcounter | Start macro definition. If pressed while defining a macro, insert a counter. |
| <Ctrl+X> ) | kmacro-end-macro | End macro definition. |
| <F4> | kmacro-end-orcall-macro | End macro definition (if definition is in progress) or invoke last keyboard macro. |
| <Ctrl+X> e | kmacro-end-andcall-macro | Execute last keyboard macro defined. Can type e to repeat macro. |
| <Ctrl+X> <Ctrl+K> n | name-last-kbdmacro | Name the last macro you created (before saving it). |
| (none) | insert-kbd-macro | Insert the macro you named into a file. |
| (none) | macroname | Execute a named keyboard macro. |
| <Ctrl+X> q | kbd-macro-query | Insert a query in a macro definition. |
| <Ctrl+u> <Ctrl+X> q | (none) | Insert a recursive edit in a macro definition. |
| <Ctrl+Alt+C> | exit-recursive-edit | Exit a recursive edit. |
| <Ctrl+X> <Ctrl+K> b | kmacro-bind-tokey | Bind a macro to a key (<Ctrl+X> <Ctrl+K> 0-9 and A-Z are reserved for macro bindings). Lasts for current session only. |
| <Ctrl+X> <Ctrl+K> Space | kmacro-step-editmacro | Edit a macro while stepping through it. |
| <Ctrl+X> <Ctrl+K> l | kmacro-editlossage | Turn the last 100 keystrokes into a keyboard macro. |
| <Ctrl+X> <Ctrl+K> e | edit-kbd-macro | Edit a keyboard macro by typing <Ctrl+X> e for the last keyboard macro defined, <Alt+X> for a named macro, <Ctrl+H> l for lossage, or keystrokes for a macro bound to a key. |
| <Ctrl+X> <Ctrl+K> Enter | kmacro-editmacro | Edit the last keyboard macro. |
| <Ctrl+X> <Ctrl+K> <Ctrl+E> | kmacro-editmacro-repeat | Edit the last keyboard macro again. |
| <Ctrl+X> <Ctrl+K> <Ctrl+T> | kmacro-swap-ring | Transpose last keyboard macro with previous keyboard macro. |
| <Ctrl+X> <Ctrl+K> <Ctrl+D> | kmacro-deletering-head | Delete last keyboard macro from the macro ring. |
| <Ctrl+X> <Ctrl+K> <Ctrl+P> | kmacro-cycle-ringprevious | Move to the previous macro in the macro ring. |
| <Ctrl+X> <Ctrl+K> <Ctrl+N> | kmacro-cycle-ringnext | Move to the next macro in the macro ring. |
| <Ctrl+X> <Ctrl+K> <Ctrl+R> | apply-macro-toregion-lines | Apply this macro to each line in a region. |

Table W25.14 Keystroke Macros

*W25.3.13.1 Using the Options Menu*

The easiest and quickest way to customize the behavior of emacs is by using the GNU emacs pull-down menu choices under Options, which is shown in Figure W25.19. For example, if you put a check mark next to the **Highlight Matching Parentheses** choice, all matching left and right parentheses in the buffer will be highlighted as you type them.

This option will only be true for the current session of emacs. If you want to retain this option for all future sessions of emacs, make the Options menu choice **Save Options**. The following valid line of elisp will automatically be written to your **~/.emacs** file, under the custom-set-variables group as seen in the following line:

**`(show-paren-mode t)**

In-Chapter Exercise

24. Show the emacs Help facility keystroke sequence you would use to find out what the option show-paren-mode is. Then, list the first few lines of how the Help facility describes the show-paren-mode option.

You can also customize by group from the **Options** menu, if you make the **Customize Emacs** choice, and then make any of the subchoices below that. For example, if you make the **Options>Customize Emacs>Top-Level Customization Group** choice, a new buffer opens on screen, and allows you to select from all of the subgroups of custom variables.

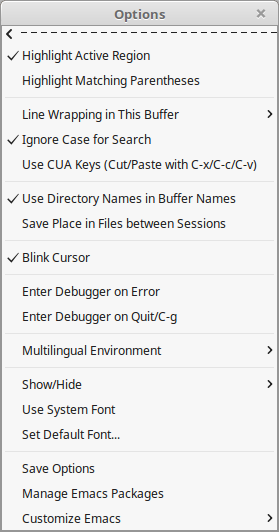


Figure W25.19 Options menu choices

The next section shows how to achieve this kind of customization as a typed command.

*W25.3.13.2 Changing Emacs Variables with Custom and the* <Alt+X> customize *Command*

Emacs has many settings that you can change. Most settings are customizable via affecting the settings of *variables*, which are also called *user options*. There is a huge number of customizable variables, controlling numerous aspects of emacs behavior. A separate class of settings, which we do not cover here, are the *faces*, which determine the fonts, colors, and other attributes of text.

To browse and alter settings (both variables and faces), at the emacs command prompt type <Alt+Shift+X> customize. This creates a customization buffer, which lets you navigate through a logically organized list of settings, edit and set their values, and save them permanently.

Customization settings are organized into *customization groups*. These groups are collected into bigger groups, all the way up to a master group called Emacs, shown near the top of the buffer in Figure W25.20.

<Alt+Shift+X> customize creates a *customization buffer* that looks similar to Figure W25.20.

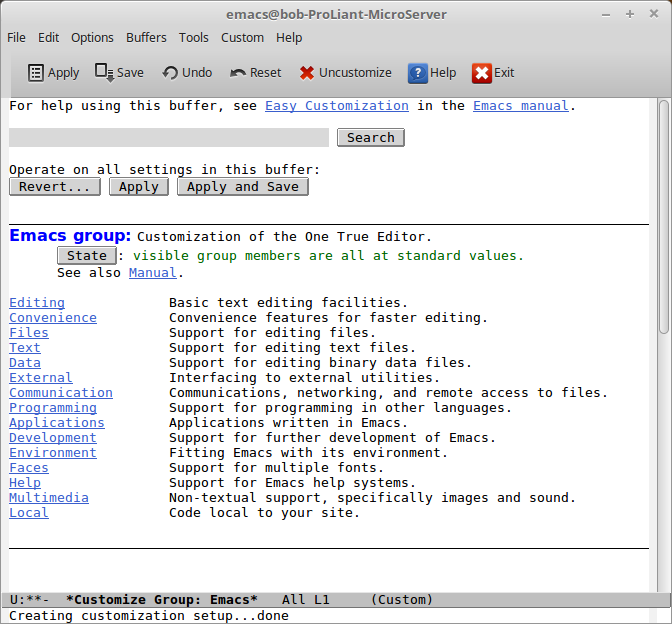


Figure W25.20 Emacs customization groups

If you are interested in customizing a particular setting or customization group that you already know the name of, you can go straight there with the commands <Alt+X> customize-option, <Alt+X> customize-face, or <Alt+X> customize-group.

The main part of the buffer in Figure W25.20 shows the “Emacs” customization group, which contains several other subgroups (“Editing,” “Convenience,” etc.). The contents of those subgroups are shown in the single line of description for each group.

The *state* of the group indicates whether the settings in that group have been edited, set or saved.

Most of the customization buffer cannot be changed, but it includes many editable fields. For example, at the top of the customization buffer is an editable field for searching for settings, with a Search button next to it. There are also buttons and links that you can activate by either clicking with the mouse, or moving the point there and then pressing <Enter>. For example, group names like “[Editing]” are links; activating one of these links brings up another customization buffer for that group.

In any particular customization buffer, you can type <Tab> (widget-forward) to move forward to the next button or editable field. <Shift+Tab> (widget-backward) moves back to the previous button or editable field.

W25.3.13.2.1 Browsing and Searching for Settings: From the top-level customization buffer created by <Alt+X> customize, you can follow the links to the subgroups of the “Emacs” customization group. These subgroups may contain settings for you to customize; they may also contain further subgroups, dealing with yet more specialized subsystems of emacs. As you graphically navigate the hierarchy of customization groups, you will find some settings that you want to customize according to your own personal preferences, and according to the nature of the text documents that you are efficiently trying to edit.

W25.3.13.2.2 Changing a Variable: Here is an example of what a variable, or user option, looks like in a specific customization buffer. This variable is accessed by descending down from the top emacs group through the groups **Editing>Killing**, and then left-clicking on the small diamond shape pointing towards the text Kill Ring Max:

Kill Ring Max: 60

[State]: STANDARD.

Maximum length of kill ring before oldest elements are thrown away.

The first line shows that the variable is named kill-ring-max, formatted as Kill Ring Max for easier viewing. Its value is 60. On our graphical display, the line after the variable name indicates the customization state of the variable: in this example, STANDARD means you have not changed the variable, so its value is the default one. The [State] button gives a menu of operations for customizing the variable.

Below the customization state is the documentation for the variable. To enter a new value for Kill Ring Max, just click to the right of the value and edit it. As you begin to alter the text, the [State] line will change to:

[State]: EDITED, shown value does not take effect until you set or save it.

Editing the value does not make it take effect right away. To do that, you must set the

variable by left-clicking on the [State] button and choosing **Set for Current Session**. Then the variable’s state becomes:

[State]: SET for current session only.

At this point, you could have made the menu choice **Save for Future Sessions**.

Also, you don’t have to worry about specifying a value that is not valid; the **Set for Current Session** operation checks for validity and will not install an unacceptable value.

When you set a variable, the new value takes effect only in the current emacs session. To save the value for future sessions, use the [State] button and select the **Save for Future Sessions** operation. Saving custom settings works by writing elisp code to a file, in this case your **~/.emacs** file. Future emacs sessions automatically read this file at startup, which invokes and establishes the customizations again.

You can also restore the variable to its standard value by using the [State] button and selecting the **Erase Customization** menu choice. There are four reset operations as follows:

* Undo edits: If you have modified but not yet set the variable, this restores the text in the customization buffer to match the actual value.
* Reset to saved: This restores the value of the variable to the last saved value, and updates the text accordingly.
* Erase customization: This sets the variable to its standard value. Any saved value that you have is also erased.
* Set to backup value: This sets the variable to a previous value that was set in the customization buffer in this session. If you customize a variable and then reset it, which discards the customized value, you can get the discarded value back again with this operation.

Sometimes it is useful to record a comment about a specific customization. Use the Add Comment item from the [State] menu to create a field for entering the comment.

W25.3.13.2.3 Globally Saving Customizations for a Group: Near the top of any group’s customization buffer, you can save all customization settings shown in that group buffer by choosing either the [Apply] or [Apply and Save] buttons. [Apply] only saves for the current session, and [Apply and Save] saves for future sessions by modifying the **~/.emacs** file accordingly by putting elisp code in the **~/.emacs** file.

W25.3.13.2.4 More about Emacs Variables: A *variable* is an elisp symbol that has a value. The symbol’s name is the *variable name*. A variable name can contain any characters that can appear in a file, but most variable names consist of ordinary words separated by hyphens.

The name of the variable is descriptive of its role in the emacs environment. Most variables also have a documentation string, which describes what the variable’s purpose is, what kind of value it should have, and how the value will be used.

You can view the documentation for a variable, such as **somevariablename**, using the help command **<Ctrl+H> v** *Describe variable:* **somevariablename** inthe minibuffer. To use this facility, type in the command **<Ctrl+H> v**; the system prompts you in the minibuffer with *Describe variable:*; then type in the variable name, such as **somevariablename**, and press **<Enter>**.

Elisp uses variables for internal record keeping, but as noted earlier, the most interesting variables for a user who will not be writing elisp programs per se are those meant for users to change—these are called *customizable variables* or *user options*.

Elisp allows any variable (with a few exceptions) to have any kind of value. However, many variables are meaningful only if assigned values of a certain type. For example, as shown in Section W25.3.13.2.2, only numbers are meaningful values for kill-ring-max, which specifies the maximum length of the kill ring; if you assign kill-ring-max a text string as a value, commands such as <Ctrl+Y> (yank) will signal an error. On the other hand, some variables don’t care about what kind or type of value you assign them; for instance, if a variable has one effect for nil values and another effect for non-nil values, then any value that is not the symbol nil induces the second effect, regardless of its type (by convention, we usually use the value t—a symbol that stands for “true”—to specify a non-nil value). If you set a variable using the customization buffer, you need not worry about giving it an invalid type: the customization buffer usually only allows you to enter meaningful values. When in doubt, use **<Ctrl+H> v** *Describe variable:* **somevariablename** to check the variable’s documentation string to see the kind of value it expects.

W25.3.13.2.5 Examining and Setting Variables: The following are some examples of how to examine and set the values of user options. The first general form of this syntax is:

**<Ctrl+H> v variablename <Enter>**

This general form uses emacs help function with the v option and displays the value and documentation for variable variablename.

The second general form achieves the change in the variables value:

**<Alt+X> set-variable <ENTER> var <ENTER> value <ENTER>**

This changes the value of variable var to value.

It reads a variable name that you supply by typing in the minibuffer, with completion, and displays both the value and the documentation of the variable. For example:

**<Ctrl+H> v fill-column <ENTER>**

A new buffer opens and displays the following:

fill-column is a variable defined in 'C source code'.

Its value is 70

Automatically becomes buffer-local when set.

This variable is safe as a file local variable if its value

satisfies the predicate 'integerp'.

Documentation:

Column beyond which automatic line-wrapping should happen.

Interactively, you can set the local value with <Ctrl+X> f

You can customize this variable.

Click on the underlined text customizeand you can use a buffer to change the value of this variable.

The most convenient keystroke method to set a specific customizable variable is by typing <Alt+X> set-variable. This reads the variable name with the minibuffer (with completion), and then reads an elisp expression for the new value that you type in the minibuffer a second time (you can insert the old value into the minibuffer for editing via <Alt+N>). For example:

**<Alt+X> set-variable <ENTER> fill-column <ENTER> 75 <ENTER>**

sets fill-column to 75.

<Alt+X> set-variable is limited to user options, customizable variables, but you can set any variable with an elisp expression like this:

**(setq fill-column 75)**

To execute such an expression, type <Alt+:> (eval-expression) and enter the expression in the minibuffer. Alternatively, go to the \*scratch\* buffer, type in the expression, and then type <Ctrl+J>.

Setting variables this way affects only the current emacs session. The only way to alter the variable for future sessions is to put the alteration as a Lisp statement in your initialization file.

*W25.3.13.3 Init File Syntax*

Your GNU emacs system’s *init* file, **~/.emacs**, contains elisp expressions. Each elisp expression consists of a function name followed by arguments, all surrounded by parentheses. For example:

**(setq fill-column 60)**

calls the function setq to set the variable fill-column to 60.

You can set any Lisp variable with setq, but with certain variables setq won’t work.

The second argument to setq is an expression for the new value of the variable. This

can be a constant, a variable, or a function call expression. In your **~/.emacs** file, constants are used most of the time. They can be one any of the following:

\* Numbers: Numbers are written in decimal, with an optional initial minus sign.

\* Strings: Lisp string syntax is the same as C string syntax with a few extra features. Use

a double-quote character (") to begin and end a string constant.

\* Characters: Lisp character constant syntax consists of a ? followed by either a character or an escape sequence starting with \.

\* True: t stands for “true.”

\* False: nil stands for “false.”

\* Other Lisp objects: Write a single quote (') followed by the Lisp object you want.

*W25.3.13.4 Keystroke Abbreviations or Abbrevs*

Just like an ordinary language abbreviation, an *abbrev* is a word which expands, when you insert it, into a usually expanded or enlarged string of text. Abbrevs are defined by the user to expand in specific ways. For example, you might define Bob as an abbrev expanding to Better off built. Then you could insert Better off built into the buffer by typing Bob <Space>.

A second kind of abbreviation facility, which we do not show examples of here, is called *dynamic abbrev expansion*. You use dynamic abbrev expansion with an explicit command to expand the letters in the buffer before the point by looking for other words in the buffer that start with those letters.

Abbrevs expand only when *Abbrev mode*, a *buffer-local minor mode*, is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not

expand until Abbrev mode is enabled again. The command <Alt+X> abbrev-mode toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise.

You can define abbrevs interactively during the editing session, irrespective of whether

Abbrev mode is enabled. You can also save lists of abbrev definitions in files, which you

can then reload for use in later sessions.

W25.3.13.4.1Defining Abbrevs: The following are ways of defining and managing abbrevs:

**<Ctrl+X> a g**

Define an abbrev, using one or more words before point as its expansion (add-global-abbrev).

**<Ctrl+X> a l**

Similar, but define an abbrev specific to the current major mode (add-mode-abbrev).

**<Ctrl+X> a i g**

Define a word in the buffer as an abbrev (inverse-add-global-abbrev).

**<Ctrl+X> a i l**

Define a word in the buffer as a mode-specific abbrev (inverse-add-mode-abbrev).

**<Alt+X> define-global-abbrev <Enter> abbrev <Enter> expression <Enter>**

Define abbrev as an abbrev expanding into an expression.

**<Alt+X> define-mode-abbrev <Enter> abbrev <Enter> expression <Enter>**

Define abbrev as a mode-specific abbrev expanding into an expression.

**<Alt+X> kill-all-abbrevs**

Discard all abbrev definitions, leaving a blank slate.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position the point after it, and type <Ctrl+X> a g. This reads the abbrev itself using the minibuffer, and then defines it as an abbrev for one or more words before the point. As with many other emacs commands, you can use a numeric digit argument to specify how many words before the point should be taken as the expansion. For example, to define the abbrev Bob, insert the text Better off built and then type <Ctrl+U> 3 <Ctrl+X> a g Bob <Enter>.

An argument of zero to <Ctrl+X> a g means to use the contents of the region as the expansion of the abbrev being defined.

To remove an abbrev definition, give a negative argument to the abbrev definition command, as in one of the following ways:

**<Ctrl+U> - <Ctrl+X> a g**

**<Ctrl+U> - <Ctrl+X> a l**

The first way removes a global definition, while the second way removes a mode-specific definition.

<Alt+X> kill-all-abbrevs removes all abbrev definitions, both global and local.

W25.3.13.4.2Controlling Abbrev Expansion: When Abbrev mode is enabled, an abbrev expands whenever it is present in the buffer just before the point and when you type a self-inserting whitespace or punctuation character like <Space> or a comma, etc. More precisely, any character that is not a word constituent expands an abbrev, and any word constituent character can be part of an abbrev. The most common way to use an abbrev is to insert it and then insert a punctuation or whitespace character to expand it.

These commands are used to control abbrev expansion:

**<Alt+'>**

Separate a prefix from a following abbrev to be expanded (abbrev-prefixmark).

**<Ctrl+X> a e**

Expand the abbrev before the point (expand-abbrev). This is effective even when Abbrev mode is not enabled.

**<Alt+X> expand-region-abbrevs**

Expand some or all abbrevs found in the region.

If you expand an abbrev by mistake, you can undo the expansion by typing C-/ (undo). This undoes the insertion of the abbrev expansion and brings back the abbrev text. You can also use the command <Alt+X> unexpand-abbrev to cancel the last expansion without deleting the terminating character.

W25.3.13.4.3Listing and Editing Abbrevs:

**<Alt+X> list-abbrevs**

Display a list of all abbrev definitions. With a numeric argument, list only local abbrevs.

<Alt+X> edit-abbrevs allows you to add, change or kill abbrev definitions by editing a list of them in an emacs buffer. The buffer of abbrevs is called **\*Abbrevs\***, and is in **Edit>Abbrevs mode**. Type <Ctrl+C> <Ctrl+C> in this buffer to install the abbrev definitions as specified in the buffer, and delete any abbrev definitions not listed.

The commands edit-abbrevs and list-abbrevs are the same except they display the listing in a window and a buffer, respectively.

W25.3.13.4.5Saving Abbrevs: These commands allow you to keep abbrev definitions between editing sessions:

**<Alt+X> write-abbrev-file <Enter> filename <Enter>**

Save to filename describing all defined abbrevs.

**<Alt+X> read-abbrev-file <Enter> filename <Enter>**

Read from **filename** and define abbrevs as specified in that file.

**<Alt+X> define-abbrevs**

Define abbrevs from definitions in current buffer.

**<Alt+X> insert-abbrevs**

Insert all abbrevs and their expansions into current buffer.

<Alt+X> write-abbrev-file reads a file name using the minibuffer and then writes a description of all current abbrev definitions into that file. This is used to save abbrev definitions for use in a later session. The text stored in the file is a series of Lisp expressions that, when executed, define the same abbrevs that you currently have.

<Alt+X> read-abbrev-file reads a file name using the minibuffer and then reads the file, defining abbrevs according to the contents of the file. The function quietly-read-abbrev-file is similar except that it does not display a message in the echo area; you cannot invoke it interactively, and it is used primarily in your init file. If either of these functions is called with nil as the argument, it uses the file given by the variable abbrev-file-name, which is **~/.emacs.d/abbrev\_defs** by default. This is your standard abbrev definition file, and Emacs loads abbrevs from it automatically when it starts up.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for <Ctrl+X> s or <Ctrl+X> <Ctrl+C >). It saves them in the file specified by abbrev-file-name. This feature can be inhibited by setting the variable save-abbrevs to nil.

The commands <Alt+X> insert-abbrevs and <Alt+X> define-abbrevs are similar to the previous commands but work on text in an emacs buffer. <Alt+X> insert-abbrevs inserts text into the current buffer after the point, describing all current abbrev definitions; <Alt+X> define-abbrevs parses the entire current buffer and defines abbrevs accordingly.

*W25.3.13.5 Keystroke Macro Commands*

Similar to the brief introduction in Section W25.3.5, in this section we more fully describe how to record, save, edit, and list a sequence of commands in a *macro*, so you can repeat it conveniently later. A keyboard macro is a command defined by an emacs user that represents, in a shortened form, another sequence of keys. For example, if you discover that you are about to type three different keystroke combinations 400 times, you can speed your work by defining a much shorter keyboard macro to do those three different keystroke combinations and then executing it 399 more times.

You define a keyboard macro by executing and recording the commands which are its definition. As you define a keyboard macro, the definition is being executed for the first time. When you close the definition, the keyboard macro is defined and also has been executed once. You can then repeat the commands by invoking the macro as many times as you like.

W25.3.13.5.1Keystroke Macros: Basic Use

These are the basic operations in defining and using keystroke macros:

**<F3>**

Start defining a keyboard macro (kmacro-start-macro-or-insert-counter).

**<F4>**

Dual-purpose function key. If a keyboard macro is being defined, end the definition; otherwise, execute the most recent keyboard macro (kmacro-end-or-call-macro).

**<Ctrl+U> <F3>**

Re-execute last keyboard macro, then append keys to its definition.

**<Ctrl+U> <Ctrl+U> <F3>**

Append keys to the last keyboard macro without re-executing it.

**<Ctrl+X> <Ctrl+K> r**

Run the last keyboard macro on each line that begins in the region (apply-macro-to-region-lines).

To start defining a keyboard macro, type <F3>. From then on, your keys continue to be executed, but also become part of the definition of the macro. Def appears in the mode line. When you are finished, type <F4> (kmacro-end-or-call-macro) to terminate the definition. For example:

**<F3> <Alt+F> Mansoor <F4>**

defines a macro to move forward a word and then insert Mansoor at the point. <F3> and <F4> do not become part of the macro.

After defining the macro, it is the most recently defined keyboard macro, and you can call it with <F4>. In the example, this has the same effect as typing <Alt+F> Mansoor again.

The two roles of the <F4> command: it ends the macro if you are in the process of defining one, or calls the last macro otherwise.

You can also supply <F4> with a numeric prefix argument n, which means to invoke the macro n times. An argument of zero repeats the macro indefinitely, until it gets an error or you type <Ctrl+G> to terminate it.

After ending the definition of a keyboard macro, you can append more keystrokes to its definition by typing <Ctrl+U> <F3>. This is equivalent to plain <F3> followed by retyping the whole definition so far. As a consequence, it re-executes the macro as previously defined. If you change the variable kmacro-execute-before-append to nil, the existing macro will not be re-executed before appending to it (the default is t). You can also add to the end of the definition of the last keyboard macro without re-executing it by typing <Ctrl+U> <Ctrl+U> <F3>.

When a command reads an argument with the minibuffer, your minibuffer input becomes

part of the macro along with the command. So when you replay the macro, the command

gets the same argument as when you entered the macro. For example:

**<F3> <Ctrl+A> <Ctrl+K> <Ctrl+X> b Mansoor <Enter> <Ctrl+Y> <Ctrl+X> b <Enter> <F4>**

defines a macro that kills the current line, yanks it into the buffer **Mansoor**, then returns to the original buffer. The command **<Ctrl+X> <Ctrl+K> r** (apply-macro-to-region-lines) repeats the last defined keyboard macro on each line that begins in the region. It does this line by line, by moving the point to the beginning of the line and then executing the macro.

All defined keyboard macros are recorded in the *keyboard macro ring*. There is only one keyboard macro ring, shared by all buffers. The basic keyboard macro ring operations are:

**<Ctrl+X> <Ctrl+K> <Ctrl+K>**

Execute the keyboard macro at the head of the ring (kmacro-end-or-callmacro-repeat).

**<Ctrl+X> <Ctrl+K> <Ctrl+N>**

Rotate the keyboard macro ring to the next macro (defined earlier) (kmacrocycle-ring-next).

**<Ctrl+X> <Ctrl+K> <Ctrl+P>**

Rotate the keyboard macro ring to the previous macro (defined later) (kmacrocycle-ring-previous).

Note: The maximum number of macros stored in the keyboard macro ring is determined by the customizable variable kmacro-ring-max.

W25.3.13.5.2 Naming, Saving, and Invoking or Using Keyboard Macros

The following are the ways to name, save, and invoke or use keyboard macros, particularly with regard to retaining them in your **~/.emacs** so that they will be available in all future sessions of emacs (anything below enclosed in [ ] is optional).

1. **<Ctrl+X> <Ctrl+K> n** **<Enter>** **macroname <Enter>**

Gives a command name (for the duration of the current emacs session only) to the most recently defined keyboard macro (kmacro-name-last-macro). If you wish to save a keyboard macro for later use, you can give it a name using this syntax. This sequence reads a name as an argument, by prompting for the name in the minibuffer, and uses the minibuffer-supplied name and defines that name so that you can execute the last keyboard macro, in its current form, using that name. The macro name is an elisp symbol, and defining it in this way makes it a valid command name for invoking or using it with <Alt+X>, or for binding a key to it with global-set-key. If you specify a name that has a prior definition other than a keyboard macro, you get an error.

2. **<Ctrl+X> <Ctrl+K> b <Enter>** **key <Enter>**

Binds the most recently defined keyboard macro to a key sequence (for the duration of the current emacs session only) (kmacro-bind-to-key).

W25. **<Alt+X> insert-kbd-macro <Enter> [macroname <Enter>]**

Inserts in the current buffer a keyboard macro’s definition as elisp code. If you do not supply an already-defined macroname, the last keyboard macro defined is inserted as elisp code.

4. **<Alt+X> macroname** **<Enter>**

Invokes macroname in the current buffer.

5. Pressing the function key <F4> invokes the last defined keyboard macro.

W25.3.13.5.3 Saving Keyboard Macros for Future Sessions

Once a keyboard macro has a name, you can save its definition in a file, and particularly in the **~/.emacs** file or other initialization file that you may use to initialize emacs at startup. By taking the following steps, it can be used in all future editing sessions.

The steps to accomplish this are as follows:

1. Visit the file you want to save the definition in, which becomes the current buffer. This is usually **~/.emacs**.

2. Use the command <Alt+X> insert-kbd-macro <Enter> macroname <Enter>

This uses the macroname you already have previously defined, and inserts equivalent elisp code that the keyboard macro represents, into the current buffer.

W25. Save the current buffer. If the file you save in is your initialization file **~/.emacs**, then the macro will be defined for all future sessions of emacs.

*The sections below describe key bindings, which map keys to commands, and keymaps, which record key bindings. They also explain how to customize key bindings, which is done by editing your emacs init file.*

W25.3.13.5.4 Keys, Commands, and Variables

Emacs does not assign meanings to keys directly. Instead, emacs assigns meanings to named commands, and then gives keys their meanings by binding them to commands.

As you have seen in the previous sections, every command has a name, which is usually made up of a few words separated by hyphens—for example, insert-kbd-macro or abbrev-file-name. Internally, each command is an emacs form of a Lisp function, and the actions associated with the command are performed by running the function.

The bindings, or mappings, between keys and commands are recorded in tables called *keymaps*.

The effect of “<Ctrl+N> moves point down vertically one line” is that the vertical movement of the command next-line is bound to the key sequence <Ctrl+N>. If you rebind <Ctrl+N> to the command forward-word, <Ctrl+N> will move forward one word instead. The key is bound to a command.

A *variable* is a name used to store a value. The variables we described in Section W25.3.13.2 are intended to be customized: some commands or mechanisms in emacs examine the variable and behave according to the value that you assign to the variable when and if you customize it.

W25.3.13.5.5 Keymaps

Emacs commands are elisp functions whose definition provides for interactive use. Like every elisp function, a command has a function name, which usually consists of lowercase letters and hyphens. A keystroke (*key* for short) sequence is a sequence of input events that have a meaning as a unit. Input events include characters, function keys, and mouse buttons—all the inputs that you can send to the computer. A key sequence gets its meaning from its binding, which dictates what command it runs.

The bindings between key sequences and command functions are recorded in data structures called *keymaps*. Emacs has many of these, each used on particular occasions.

The global keymap is the most important keymap because it is always in effect. The *global keymap* defines keys for *Fundamental mode*; most of these definitions are common to most or all major modes. Each major or minor mode can have its own keymap which overrides the global definitions of some keys.

For example, a self-inserting character such as g is self-inserting because the global keymap binds it to the command self-insert-command. The standard emacs editing characters such as <Ctrl+A> also get their standard meanings from the global keymap. Commands to rebind keys, such as <Alt+X> global-set-key, work by storing the new binding in the proper place in the global map.

Most modern keyboards have function keys as well as character keys. Function keys

send input events just as character keys do, and keymaps can have bindings for them. Key sequences can mix function keys and characters. For example, if your keyboard has a <Home> function key, emacs can recognize key sequences like <Ctrl+X> <Home>. You can even mix mouse events with keyboard events, such as S-down-mouse-1.

On text terminals, typing a function key actually sends the computer a sequence of characters; the precise details of the sequence depends on the function key and on the terminal type. (Often the sequence starts with ESC [). If emacs understands your terminal type properly, it automatically handles such sequences as single input events.

W25.3.13.5.6 Prefix Keymaps

Emacs stores only single events in each keymap. Interpreting a key sequence of multiple events involves a chain of keymaps: the first keymap gives a definition for the first event, which is another keymap that is used to look up the second event in the sequence, and so on. A *prefix key* such as <Ctrl+X> or <Esc> has its own keymap, which holds the definition for the event that immediately follows that prefix.

A prefix key is usually the keymap to use for looking up the following event. The definition can also be an elisp symbol whose function definition is the following keymap; the effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of <Ctrl+X> is the symbol Control-X-prefix, whose function definition is the keymap for <Ctrl+X> commands. The definitions of <Ctrl+C>, <Ctrl+X>, <Ctrl+H>, and <Esc> as prefix keys appear in the global map, so these prefix keys are always available.

Some prefix keymaps are stored in variables with names:

**\_ ctl-x-map** is the variable name for the map used for characters that follow <Ctrl+X>.

**\_ help-map** is for characters that follow <Ctrl+H>.

**\_ esc-map** is for characters that follow <Esc>. Thus, all metacharacters are actually defined by this map.

**\_ ctl-x-4-map** is for characters that follow <Ctrl+X> 4.

**\_ mode-specific-map** is for characters that follow <Ctrl+C>.

W25.3.13.5.7 Local Keymaps

So far, we have explained the ins and outs of the global map. Major modes customize emacs by providing their own key bindings in *local keymaps*. For example, C mode overrides <Tab> to make it indent the current line for C code. Minor modes can also have local keymaps; whenever a minor mode is in effect, the definitions in its keymap override both the major mode’s local keymap and the global keymap. In addition, portions of text in the buffer can specify their own keymaps, which override all other keymaps.

A local keymap can redefine a key as a prefix key by defining it as a prefix keymap. If the key is also defined globally as a prefix, its local and global definitions (both keymaps) effectively are combined: both definitions are used to look up the event that follows the prefix key. For example, if a local keymap defines <Ctrl+C> as a prefix keymap, and that keymap defines <Ctrl+Z> as a command, this provides a local meaning for <Ctrl+C> <Ctrl+Z>. This does not affect other sequences that start with <Ctrl+C>; if those sequences don’t have their own local bindings, their global bindings remain in effect.

Another way to think of this is that emacs handles a multievent key sequence by looking in several keymaps, one by one, for a binding of the whole key sequence. First it checks the minor mode keymaps for minor modes that are enabled, then it checks the major mode’s keymap, and then it checks the global keymap.

W25.3.13.5.8 Changing Key Bindings Interactively

The way to redefine an emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local bindings for the same key), or you can change a local keymap, which affects all buffers using the same major mode.

The following describes how to rebind keys for the current emacs session (see Section W25.3.13.5.9 for a description of how to make key rebindings affect future emacs sessions by putting them in your **~/.emacs** file):

1. **<Alt+X> global-set-key <Enter> key command <Enter>**

Defines key globally to run command.

2. **<Alt+X> local-set-key <Enter> key command <Enter>**

Defines key locally (in the major mode now in effect) to run command.

W25. **<Alt+X> global-unset-key <Enter> key**

Makes key undefined in the global map.

4. **<Alt+X> local-unset-key <Enter> key**

Makes key undefined locally (in the major mode now in effect).

For example, the following binds <Ctrl+Z> to the shell command, replacing the normal global definition of <Ctrl+Z>:

**<Alt+X> global-set-key <Enter> <Ctrl+Z> shell <Enter>**

The global-set-key command reads the command name after the key. After you press the key, a message like this appears so that you can confirm that you are binding the key you want:

*Set key <Ctrl+Z> to command:*

You can redefine function keys and mouse events in the same way; just type the function key or click the mouse when it’s time to specify the key to rebind. You can rebind a key that contains more than one event in the same way. Emacs keeps reading the key to rebind until it is a complete key (that is, not a prefix key). Thus, if you type <Ctrl+F> for the key, that’s the end; it enters the minibuffer immediately to read the command. But if you type <Ctrl+X>, since that’s a prefix, it reads another character; if that is 4, another prefix character, it reads one more character, and so on. For example:

**<Alt+X> global-set-key <Enter> <Ctrl+X> 4 $ spell-other-window <Enter>**

redefines <Ctrl+X> 4 $ to run the (fictitious) command spell-other-window.

You can remove the global definition of a key with global-unset-key. This makes the key undefined; if you type it, emacs will just beep. Similarly, local-unset-key makes a key undefined in the current major mode keymap, which makes the global definition (or lack of one) come back into effect in that major mode.

If you have redefined (or undefined) a key and you subsequently wish to retract the change, undefining the key will not do the job; you need to redefine the key with its standard definition.

To find the name of the standard definition of a key, go to a Fundamental mode buffer in an emacs session that you have not done any key remappings in, and type <Ctrl+H> c. So, if you want to prevent yourself from invoking a command by mistake, it is better to disable the command than to undefine the key!

W25.3.13.5.9 Rebinding Keys in Your Init File

If you have a set of key bindings that you like to use all the time, you can specify them in your initialization file by writing elisp code. There are several ways to write a key binding using elisp. The simplest is to use the kbd function, which converts a text representation of a key sequence, similar to how we have written key sequences up to this point, into a form that can be passed as an argument to global-set-key. For example, here’s how to bind <Ctrl+Z> to the shell command.

**(global-set-key (kbd "C-z") 'shell)**

The single-quote (') before the shell command name designates it as a constant symbol rather than a variable. If you omit the quote, emacs tries to evaluate shell as a variable.

W25.3.13.5.10 Examples

Here are some additional examples, including binding function keys and mouse events:

**(global-set-key (kbd "<Ctrl+C> y") 'clipboard-yank)**

**(global-set-key (kbd "<Ctrl ><Alt+Q>") 'query-replace)**

**(global-set-key (kbd "<f5>") 'flyspell-mode)**

**(global-set-key (kbd "<Ctrl ><f5>") 'linum-mode)**

**(global-set-key (kbd "<Ctrl ><right>") 'forward-sentence)**

**(global-set-key (kbd "<mouse-2>") 'mouse-save-then-kill)**

In-Chapter Exercises

25. (a) Use the emacs Help function via keyboard keystrokes only to find out what the commands that are being bound to each of the keys sequences in the six examples in Section 3.13.5.10 accomplish. So, for forward-sentence, what explanation does Help supply? Make a list of the answers that the Help function supplies.

(b) What are the default key sequence bindings, if any, for the commands in the six examples? Make a list of the default key sequence bindings for commands that have them.

26. Place all six examples of key sequences bound to commands in your **~/.emacs** file and test them according to your findings in In-Chapter Exercise 25..

W25.4 SUMMARY

In this chapter, we covered vi/vim/gvim and emacs, the most useful group of text editors that Linux offers. We achieved this in both a command line, text-based way, and in a graphical way, for these editors. They are useful because modern Linux is both a text-driven and GUI-based operating system. Common operations done by an ordinary user, such as editing script files, writing e-mail messages, or creating C language programs, are done with text editors. A full-screen display editor shows a portion of a file that fills most or all of the screen display. The cursor, or point, can be moved to any of the text shown in the screen display. Editing a file involves editing a copy that the editor creates, called a buffer. Keystroke commands are one of the primary ways of interacting with these editors. Using a GUI to interact with these editor is time efficient and easy to learn. The editor(s) used should fit the user’s personal criteria, particularly if the user is new to Linux.

The most important functions that are common to these Linux text editors are cursor movement, cut/copy and paste, deleting text, inserting text, opening an existing file, starting a new file, quitting, saving, and search and replace.

We provide summary tables of commands and operations for vi/vim and Emacs as Tables W25.8 and W25.9.

QUESTIONS AND PROBLEMS

*vi, vim, gvim*

1. Despite the availability of fancy and powerful word processors, why is text editing still important?

2. List 10 commonly used text-editing operations.

3. What are the four most popular text editors in Linux ? Which one is your favorite? Why?

4. What is an editor buffer in vi, vim, and gvim?

5. This problem assumes you are using the Bash shell in Linux (the default shell), and will execute the file you will create in your home directory on the system.

a. Make sure that your search path includes the directory you are saving the following script file to, and that you have execute privileges on the file.

Use vi or vim on your system and create a bash shell script file that contains the lines:

**#!/bin/bash**

**echo $SHELL**

**cat /etc/shells**

Then save the file as **sheller** and quit vi. At the Bash shell prompt, type ./sheller and then press <Enter>.

b. What appears on your screen? In particular, what shells are available?

6. Run vi on your Linux system. Create and edit a block of text that you want to be the body of an e-mail message explaining the basic capabilities of the vi editor. For example, part of your message might describe the difference between the Insert and Command modes. This file should be at least one page (45 to 50 lines of text) long. Then save the file as **vi\_doc.txt**. Insert the body of text in an e-mail message and send it to yourself.

7. Run vi on your system and create a file of definitions in your own words, without looking at the textbook, for:

(a) full-screen display editor

(b) modeless editor

(c) file versus buffer

(d) keystroke commands

(e) substitute versus search

(f) text file versus binary file

Then refer back to the relevant sections of this chapter to check your definitions. Make any necessary corrections or additions. Re-edit the file in vim to incorporate any corrections or additions that you made, and then print out the file using the print commands available on your system.

8. Edit the file you created in Problem 7., and change the order of the text of your definitions to (d), (a), (c), and (b), using the yank, put, and D or dd commands. Print out the file using the print commands available on your Linux system.

1. 9. Log on to your Linux system, and execute the **vi** program on a new, blank file.
   1. On the first line of the file, type your first and last name.
   2. On the second line of the file, type " The Linux vi text editor has almost all the features of a word processor and tremendous flexibility in creating text files".
   3. Print the file to your Linux system line printer while you are still in **vi**. How do you accomplish this, in a non-GUI environment?
2. 10. What **vi** command allows you to move to the first line in the current buffer? What command allows you to move to the last line in the buffer?
3. 11. Use the **set** command to force **vi** into a 30 column by 15 line display of characters so that one screen of the display shows only 15 lines, and text is automatically wrapped onto the next line after the 30th character. How did you do this? (HINT: The **set all** command shows the current status of all **vi** environment variables.)
4. *Advanced vi,vim,gvim*
5. 12. In Section W25.2.7 above, you changed the behavior of vi, vim, and gvim by adding or modifying entries in your ~/.exrc or ~/.vimrc files, so that the changes were persistent across all sessions of the editors. You can also customize vi, vim, and gvim by changing the shell environment variable named EXINIT. This can be achieved in the C shell by giving value(s) to the **SETENV** variable. Do the following-
7. a) Refer to Chapter W29 for the C shell to find the exact syntax and use of the **SETENV** command in the C shell. Then, add or modify the shell environment variable setting for the shell variable **EXINIT** so that the **showmode** user option is turned on. What is the syntax of the command you used to do this?
8. b) How would you test that this environment variable is actually implementing the user option change, and not what is in the ~/.exrc or ~/.vimrc files?
9. c) What syntax would you use for the **setenv** command to change more than one user option in the editors?
10. d) Are these changes in the **EXINIT** variable persistent through all vi, vim, and gvim sessions? If you log out and log back into the system, does **EXINIT** still contain the changes and additions you make to it? Why, or why not?
11. 13. Give the exact syntax of a vi **substitute** command line that only replaces every instance of the discrete word ate on all the lines of a file, with the word ion, where the file has some words that end in the string ate.
12. 14. Give the exact syntax of a vim **substitute** command line that interactively searches and substitutes the word cool for the word cold on all the lines of a file, where there are several widely-separated instances of the word cold in the file.
13. 15. Take the following **map** command for creating a skeleton C program template, and place it in your ~/.exrc file-
14. **map #3 ^[i#include stdio.h ^Mmain(argc, argv) ^M int argc;^M char #argv[];^M{^M}^M^[**
15. where:
16. **^[** stands for pressing **<Ctrl>+V** and then **<Esc>**
17. **^M** stands for pressing **<Ctrl>+V** and then the **<Enter>** key
18. As stated in the text, the relative number of spaces in the above **map** command definition controls the indentation of the skeleton construct. Also, the **^M** entries put each of the skeleton construct components on a new line.
19. a) Make sure that the relative indentation of the header components and other parts of the skeleton is correct.
20. b) Add an #include <stderr.h> , #include <stdlib.h> , and #include <string.h> as header information to the skeleton.
21. c) Run the **map** command in a blank vi buffer and test it.
22. 16. To practice with the command line window to re-use any previous searches in your search history, use vim to edit the file multiline you created in Practice Session W25.4 above. Then search for the words engineers, system administrators, web servers, scientists, networking, and mathematicians, one search at a time starting at the first character in the buffer. Open a command line window on your search history. Modify the search commands in the history of searches for scientists, networking, and mathematicians to be commands that substitute the words people, homeserver gurus, and UNIX students for the words scientists, networking, and mathematicians. Save the file.
23. *GNU emacs*
24. 17. This problem assumes that –
25. You can interactively start up a new non-login shell, the Bourne Again shell, or *bash*, which is already installed by default on that system. To do this, at the Bash shell prompt just type **bash** and press <Enter>. If a ~/.bashrc exists, before you begin, be sure to back up your existing ~/.bashrc file by using the **cp** command. To do so, type- **cp .bashrc .bashrc\_bak** and then press <Enter>. If for any reason you destroy the contents of the **~/**.bashrc file while doing this problem, you can restore the original by typing- **cp .bashrc\_bak .bashrc** and then pressing <Enter>.
26. If there is no .bashrc file in your home directory, use emacs to create one and save it in your home directory as an empty file (with nothing in it). Also, type
27. **chmod u+x .bashrc** and press <Enter>.
28. Use emacs to edit the ~/.bashrc file in your home directory, and then use the **<Ctrl-X>I** command to insert the file alien3 that you created in Practice Session W25.12 into the buffer. Save the buffer, exit emacs, and log off your computer system. Log on to your computer system again, start up a new bash shell interactively by typing **bash** at the command line (so that the new ~/.bashrc is in effect), and test each of the DOS aliases that are in alien3 by typing them at the shell prompt, with their proper arguments (if necessary). They should give you the same results as when you ran the Bourne shell aliases in Step 21 of Practice Session W25.12.
29. What other way can you invoke the ~/.bashrc file immediately in this interactive session without logging off the system?
30. 18. As you saw in the Practice Sessions, you can be editing more than one file at a time in emacs, where each of the files’ contents are being held in different buffers. Experiment by first using the cp command at the shell prompt to make a copy of the file datafile that you created in Practice Session W25.9. Name this copy datafile2. Use emacs to open both files, datafile and datafile2, with the command **<Ctrl-X><Ctrl-F>** You can switch between buffers with **<Ctrl-X B>**. Then edit both of them at the same time and cut and paste three or four lines of each between the two, using **<Ctrl-@>**, **<Ctrl-W>**, and **<Ctrl-Y>**.
31. Don’t save your changes to the file datafile!
32. 19. Write a keyboard macro, as described in Section W25.3.13.4, to do everything shown in Steps 10–16 of Practice Session W25.10.
33. 20. Try working with emacs in a text-only window, and use only keystroke commands.
34. To do this, you will have to launch emacs from a console or terminal window by typing **emacs -nw newfile** The **-nw** option specifies that emacs will run in a nongraphical mode. Then, in the console or terminal window, a non-graphical emacs will open on the buffer newfile. Remember that you can still gain access to the Menu Bar menus at the top of the emacs screen by pressing the escape **<Esc>** key on the keyboard and then pressing the single backquote (**`**) key. You can then descend through the menu bar choices by pressing the letter key of the menu choice you want to make. For example, pressing the **F** key on the keyboard gives you access to the File pull-down menu choices, and then pressing the **S** key on the keyboard allows you to save the current buffer.
35. 21. To compare keystroke to graphical emacs, repeat Problem 20., using purely graphical emacs—that is, with no keystroke commands allowed. This time, make two copies of datafile named datafilex and datafilexx at the UNIX shell prompt with the **cp** command. Open all three files and, using the multiple-buffer and multiple window capability of an X Window emacs, cut and paste among the files using only the mouse. Again, as in Problem 20, don’t save your changes to the file datafile.
36. 22. Use emacs’ capability of sending e-mail while you’re in emacs. Send an e-mail message to one of your friends, composing the message body and sending from within emacs.
37. 23. Use the **<Alt-x> customize** facility in emacs to find the values of the following-
38. Global Mark Ring Max, Tab Width, Fill Column, Standard Indent, Undo Limit, and provide a list of the values you find for each.
39. 24. What emacs command toggles Abbrev mode? What emacs command removes all abbrev definitions, including global ones?
40. 25. Define the following abbreviation as global abbreviations in emacs with Abbrev using the word on the left of the equal sign (=) as the abbreviation, and list the command and keystrokes you used to create the abbreviations, and invoke them-
41. **now = Now is the time for all good women to come to the aid of their country.**
42. 26. Define a GNU emacs keyboard macro that, when invoked, automatically enters all 26 lower-case letters of the alphabet, with a single space between each letter, at point. Name the macro **le** and bind it to the key **1** (the numeric number 1) for use only during this session of emacs. Give the exact steps, commands, and typed-input you use to accomplish defining this macro and invoking it.
43. 27. Define a GNU emacs keyboard macro that, when invoked, automatically enters the integers 1 through 10, with a single space in between each number, at point. Name that macro **row** and bind it to the key **r** so that both the name and the key binding can be used in every subsequent emacs session. Give the exact steps, commands, and typed-input you use to accomplish defining this macro and invoking it.
44. 28. Define a line of elisp code and place it in your ~/.emacs file, that will designate the second mouse button on your mouse to issue a command to split the current buffer window horizontally.

*Advanced GNU Emacs*

1. 29. Using **emacs**, type in a paragraph of text from one of your favorite books, but without altering the size or shape of the **emacs** frame or using the Enter key, use the word wrap feature of **emacs** to format it exactly the way that it is printed in the book. Print the file at your Linux system line printer.
2. 30. Define an **emacs** keyboard macro that accomplishes a common editing task for you.
3. 31. Create, edit, compile, link and execute a short C program of your choice in **emacs**.

32. zenity is a graphucal GTK+ dialog box program that allows you to create interactive dialog boxes using Bash script files. It is installed by default on our Linux system. In this problem, you will use emacs to create a zenity Bash script file.

a. Use emacs to create and save the following bash script file, named zen1, in your home directory:

#!/bin/bash

zenity --forms --title="newusers Command" --text="Add batch new user" \

--add-entry="Username" \

--add-password="Password" \

--add-entry="User Number UID" \

--add-entry="Group Number GID" \

--add-entry="GECOS Entry" \

--add-entry="Default Home Directory" \

--add-entry="Default Shell" \

>> zen\_out

sed -i -e 's/|/:/g' /home/bob/zen\_out

b. On the command line, type ./zen1

A zenity dialog box will open on-screen. In the GUI dialog box you will create the seven fields needed to be supplied to the **newusers** command, which we show in Chapter 19, to create new users from a “batch file” on your Linux system. The seven inputs you supply to the dialog box will be written to a file named zen\_out.

The seven fields, separated by the colon character (:), are the new user accounts name, password, UID, GID, GECOS commentary, default home directory, and default shell.

For example-

hassan:QQQ:2001:2001:CFO of Accounting:/home/hassan:/bin/bash

c. Use zen1 to create a file of several new users you want to put on your system.

TABLE W25.8 Vi, Vim, Gvim Command Summary

|  |  |
| --- | --- |
| **Vi Syntax** |  |
| **Command** | **Action** |
| cw | Change word. |
| cc | Change line. |
| c$ | Change text from current position to end of line. |
| C | Same as c$. |
| dd | Delete current line. |
| 7 dd | Delete seven lines. |
| d$ | Delete text from current position to end of line. |
| D | Same as d$. |
| 5dw | Deletes five words. |
| d7,14 | Deletes lines seven through fourteen in the buffer. |
| s | Substitute character. <Esc> ends substitute mode. |
| 4s | Substitute four characters. <Esc> ends substitute mode. |
| S | Substitute entire line. <Esc> ends substitute mode. |
| u | Undo last change. |
| <Ctrl+R> | Redo last change (vim and gvim). |
| U | Restores the current line, if you have not moved off of it. |
| x | Delete current cursor position. |
| X | Delete back one character. |
| 5X | Delete previous five characters |
| . | Repeat last change. |
| ~ | Change case and move cursor right. |
| <Ctrl+A> | Increment number at the cursor (vim and gvim). |
| <Ctrl+X> | Decrement number at the cursor (vim and gvim). |
| **Vi Mode Keys** |  |
| **Key** | **Action** |
| a | Appends text after the character the cursor is on. |
| A | Appends text after the last character of the current line. |
| c | Begins a change operation, allowing you to modify text. |
| C | Changes from the cursor position to the end of the current line. |
| i | Inserts text before the character the cursor is on. |
| I | Inserts text at the beginning of the current line. |
| o | Opens a blank line below the current line and puts the cursor on that line. |
| O | Opens a blank line above the current line and puts the cursor on that line. |
| **Vi Command Mode** |  |
| **Command** | **Action** |
| :wq | Saves the buffer and quits. |
| :w | Saves the current buffer and remains in the editor. |
| :w filename | Saves the current buffer to filename |
| :q | Quit vi (fails if changes were made). |
| :q! | Quit vi without saving the buffer. |
| :Q | Quit vi and invoke ex. |
| :vi | Return to vi after Q command. |
| ZZ | Quits vi, saving the file only if changes were made since the last save. |
| **Vi Cursor Movement** |  |
| **Command** | **Action** |
| 1G | Moves the cursor to the first line of the file. |
| G | Moves the cursor to the last line of the file. |
| 0 (zero) | Moves the cursor to the first character of the current line. |
| <Ctrl+G> | Reports the position of the cursor in terms of line # and column #. |
| $ | Moves the cursor to the last character of the current line. |
| w | Moves the cursor forward one word at a time. |
| b | Moves the cursor backward one word at a time. |
| x | Deletes the character at the cursor position. |
| dd | Deletes the line at the current cursor position. |
| u | Undoes the most recent change. |
| r | Replaces the character at the current cursor location with what is typed next. |
| **Vi Yank and Put** |  |
| **Command Syntax** | **What It Accomplishes** |
| y2W | Yanks two words, starting at the current cursor position, going to the right. |
| 4yb | Yanks four words, starting at the current cursor position, going to the left. |
| yy or Y | Yanks the current line. |
| p | Puts the yanked text after the current cursor position. |
| P | Puts the yanked text before the current cursor position. |
| **Vi Substitute** |  |
| **Command Syntax** | **What It Accomplishes** |
| :s/john/jane/ | Substitutes the word jane for the word john on the current line, only once. |
| :s/john/jane/g | Substitutes the word jane for every word john on the current line. |
| **Vi Environment Options** |  |
| **Last line mode syntax** | **What it does** |
| **abbr command** |  |
| :ab in out | Use in as abbreviation for out in Insert mode. |
| :unab in | Remove abbreviation for in. |
| :ab | List abbreviations. |
| map!, map commands |  |
| :map | List character strings that are mapped. |
| :map! string sequence | Map characters string to input mode sequence. |
| :unmap! string | Remove input mode map (you may need to quote the characters with <Ctrl+V>). |
| :map! | List character strings that are mapped for input mode. |
| **set command** |  |
| :set x | Enable boolean option x, show value of other options. |
| :set | Show changed options. |
| :set all | Show all options. |
| :set x? | Show value of option x. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Emacs Commands** | |  | | |
| **Command** | | **Action** | | |
| <Ctrl+X> <Ctrl+F> | | Visit a file (find-file) | | |
| <Ctrl+X> <Ctrl+R> | | Visit a file for viewing, without allowing changes to it (find-file-read-only) | | |
| <Ctrl+X> <Ctrl+V> | | Visit a different file instead of the one visited last (find-alternate-file) | | |
| <Ctrl+X> <Ctrl+S> | | Save the current buffer to its file (save-buffer) | | |
| <Ctrl+X> s | | Save any or all buffers to their files (save-some-buffers) | | |
| <Alt+~> | | Forget that the current buffer has been changed (not-modified) | | |
| <Ctrl+X> <Ctrl+W> | | Save the current buffer with a specified file name (write-file) | | |
| <Ctrl+H> | | Display a help message about these options | | |
| <Ctrl+X> <Ctrl+C> | | Exits emacs | | |
| <Ctrl+X> <Ctrl+Z> | | Suspends emacs and exits to the shell | | |
| **Emacs Help Command** | | | | |
| <Ctrl+H> a topics <Enter> | | Display a list of commands whose names match topics (apropos-command). | | |
| <Ctrl+H> b | | Display all active key bindings—minor mode bindings first, then those of the major mode, then global bindings (describe-bindings). | | |
| <Ctrl+H> c key | | Show the name of the command that the key sequence key is bound to (describe-key-briefly). Here c stands for “character.” For more extensive information on key, use <Ctrl+H> k. | | |
| <Ctrl+H> d topics <Enter> | | Display the commands and variables whose documentation matches topics (apropos-documentation). | | |
| <Ctrl+H> e | | Display the **\*Messages\*** buffer (view-echo-area-messages). | | |
| <Ctrl+H> f function <Enter> | | Display documentation on the Lisp function named function (describe-function). Since commands are Lisp functions, this works for commands too. | | |
| <Ctrl+H> r | | Display the emacs manual in Info (info-emacs-manual). | | |
| <Ctrl+H> s | | Display the contents of the current syntax table (describe-syntax). The syntax table says which characters are opening delimiters, which are parts of words, and so on. | | |
| <Ctrl+H> t | | Enter the emacs interactive tutorial (help-with-tutorial). | | |
| <Ctrl+H> K key | | Enter Info and go to the node that documents the key sequence key (Info-goto-emacs-key-command-node). | | |
| <Ctrl+H> | | Display the help message for a special text area, if the point is in one (display-local-help). (These include, for example, links in **\*Help\*** buffers.) | | |
| **Emacs Cursor Movement** | | | | |
| **Entity to Move Over** | | | **Backward** | **Forward** |
| Character |  | | <Ctrl+B> | <Ctrl+F> |
| Word |  | | <Alt+B> | <Alt+F> |
| Line |  | | <Ctrl+P> | <Ctrl+N> |
| Go to line beginning (or end) |  | | <Ctrl+A> | <Ctrl+E> |
| Sentence |  | | <Alt+A> | <Alt+E> |
| Paragraph |  | | <Alt+{> | <Alt+}> |
| Page |  | | <Ctrl+X> [ | <Ctrl+X> ] |
| **Entity to Kill** | | | **Backward** | **Forward** |
| Character (delete, not kill) |  | | <Del> | <Ctrl+D> |
| Word |  | | <Alt+Del> | <Alt+D> |
| Line (to end of) |  | | <Alt+0> <Ctrl+K> | <Ctrl+K> |
| Sentence |  | | <Ctrl+X> DEL | <Alt+K> |
| Kill region | <Ctrl+W> | |  |  |
| Copy region to kill ring | <Alt+W> | |  |  |
| Yank back last thing killed | <Ctrl+Y> | |  |  |
| **Emacs Interactive Search and Replace** | | | | |
| **Search and Replace Action** | | | **Keystrokes** | |
| Search forward | | | <Ctrl+S> | |
| Search backward | | | <Ctrl+R> | |
| Regular expression search | | | <Ctrl+Alt+S> | |
| Reverse regular expression search | | | <Ctrl+Alt+R> | |
| Select previous search string | | | <Alt+P> | |
| Select next later search string | | | <Alt+N> | |
| Exit incremental search | | | <Enter> | |
| Undo effect of last character | | | <Del> | |
| Abort current search | | | <Ctrl+G> | |
| Interactively replace a text string | | | <Alt+%> | |
| Using regular expressions | | | <Alt+X> query-replace-regexp | |
| Replace this one, go on to next | | | <Space> or y | |
| Replace this one, don’t move | | | , | |
| Skip to next without replacing | | | <Del> or n | |
| Replace all remaining matches | | | ! | |
| Back up to the previous match | | | ^ | |
| Exit query-replace | | | <Enter> | |
| Enter recursive edit **(<Ctrl+Alt+C>** to exit) | | | <Ctrl+R> | |
| **Changing Emacs Behavior** | | | | |
| **Customization Action** | | | **Keystrokes** | |
| **Abbrevs** | | | | |
| add global abbrev | | | <Ctrl+X> a g | |
| add mode-local abbrev | | | <Ctrl+X> a l | |
| add global expansion for this abbrev | | | <Ctrl+X> a i g | |
| add mode-local expansion for this abbrev | | | <Ctrl+X> a i l | |
| explicitly expand abbrev | | | <Ctrl+X> a e | |
| expand previous word dynamically | | | <Alt+/> | |
| **Macros** | | | | |
| Start defining a keyboard macro | | | <Ctrl+X> ( or <F3> | |
| End keyboard macro definition | | | <Ctrl+X> ) or <F4> | |
| Execute last-defined keyboard macro | | | <Ctrl+X> e or <F4> | |
| Append to last keyboard macro | | | <Ctrl+U> <Ctrl+X> ( | |
| Name last keyboard macro | | | <Alt+X> name-last-kbd-macro | |
| Insert Lisp definition in buffer | | | <Alt+X> insert-kbd-macro | |
| Customize variables and faces | | | <Alt+X> customize | |

Table W25.9 Summary of emacs Commands