HARLEY HAHN'S

Emacs Field Guide

Harley Hahn

Apress®

Harley Hahn's Emacs Field Guide

Harley Hahn

Apress®

Harley Hahn's Emacs Field Guide

Harley Hahn
www.harley.com

ISBN-13 (pbk): 978-1-4842-1702-3 ISBN-13 (electronic): 978-1-4842-1703-0
DOI 10.1007/978-1-4842-1703-0

Library of Congress Control Number: 2016938804

Managing Director: Welmoed Spahr
Lead Editor: Jeffrey Pepper
Technical Reviewer: Dmitry Shkatov
Copyeditor: Lydia Hearn
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Mark Powers
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com/9781484217023. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

To Maria, for love and support.

And to Sadie (my dog) and Max (Maria's dog),
for being such good company.

Contents at a Glance

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

A Personal Note from Harley Hahn xxi

■ Chapter 1: All About Emacs 1

■ Chapter 2: Unix for Emacs Users 13

■ Chapter 3: Installing Emacs 57

■ Chapter 4: The Emacs Keyboard 71

■ Chapter 5: Starting and Stopping Emacs 79

■ Chapter 6: Commands, Buffers, Windows 87

■ Chapter 7: The Text Editing Work Environment 105

■ Chapter 8: The Cursor; Line Numbers; Point and Mark; The Region 121

■ Chapter 9: Kill and Delete; Move and Copy; Correct Mistakes; Spelling; Fill 137

■ Chapter 10: Searching 155

■ Chapter 11: Modes; Customizing Using Your `.emacs` File 173

Chapter 12: Shell Commands; Help and Info; Programs and Games 189

Appendix A: Personal Notes 215

Appendix B: Command Summaries 231

Index of Emacs Key Sequences 257

Index of Emacs Variables and Functions 261

Index of Unix Keys, Files and Commands 263

General Index 265

Contents

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

A Personal Note from Harley Hahn xxi

Chapter 1: All About Emacs 1

Section 1.1: Getting Started Together 1

Section 1.2: Emacs Is a Text Editor 3

Section 1.3: Emacs Is a Working Environment 5

Section: 1.4: Where Did Emacs Come From? 8

Section 1.5: The Free Software Foundation 9

Section 1.6: Excerpts From *The Gnu Manifesto* 10

Chapter 2: Unix for Emacs Users 13

Section 2.1: Operating Systems 13

Section 2.2: Unix and Linux 14

Section 2.3: Unix Terminals and Userids 18

Section 2.4: Types of Terminals 21

Section 2.5: User Interfaces 23

Section 2.6: Using a Unix Terminal 27

Section 2.7: The Unix Command Line 31

Section 2.8: The Shell Prompt 32

Section 2.9: What Unix Commands Look Like 33
Section 2.10: Making Corrections as You Type Commands 34
Section 2.11: Two Important Keys: <Ctrl-C> and <Ctrl-D> 35
Section 2.12: The History List; Command Line Editing 37
Section 2.13: The Unix Manual 40
Section 2.14: Using the `less` Pager Program 41
Section 2.15: The Three Types of Unix Files 43
Section 2.16: The Tree-Structured Filesystem 45
Section 2.17: The Current Directory and Pathnames 48
Section 2.18: File and Directory Names 52
Section 2.19: File and Directory Names: OS X and Windows 55

Chapter 3: Installing Emacs 57

Section 3.1: Installing Software: Packages vs. Manual Installation 57
Section 3.2: Installing Emacs Using a Linux Package Manager 60
Section 3.3: Installing Emacs Manually With Linux 62
Section 3.4: Installing Emacs With OS X 65
Section 3.5: Installing Emacs With Microsoft Windows 68

Chapter 4: The Emacs Keyboard 71

Section 4.1: A Strategy for Learning Emacs 71
Section 4.2: The Ctrl Key 72
Section 4.3: The Meta (Alt) Key 72
Section 4.4: Special Key Names 74
Section 4.5: The Meta Key, Bucky Bits, and Much More 75
Section 4.6: Meta Key Problems When Using a Terminal Window 78

Chapter 5: Starting and Stopping Emacs..... 79

Section 5.1: Starting Emacs 79

Section 5.2: Starting Emacs in a Terminal Window 81

Section 5.3: Starting Emacs as a Read-Only Editor 82

Section 5.4: Recovering Data After a System Failure 83

Section 5.5: Stopping Emacs..... 84

Chapter 6: Commands, Buffers, Windows 87

Section 6.1: Commands and Key Bindings..... 87

Section 6.2: Buffers..... 89

Section 6.3: Windows 91

Section 6.4: The Mode Line / Read-Only Viewing..... 94

Section 6.5: The Echo Area / Typing Emacs Commands..... 96

Section 6.6: The Minibuffer 97

Section 6.7: Completion 99

Section 6.8: Disabled Commands..... 102

Chapter 7: The Text Editing Work Environment 105

Section 7.1: How to Practice Using Emacs..... 105

Section 7.2: Typing and Correcting..... 107

Section 7.3: The `repeat` and `undo` Commands; Redo..... 108

Section 7.4: The `keyboard-quit` Command (`C-g`) 112

Section 7.5: Emacs for `vi` Users 112

Section 7.6: Commands to Control Windows..... 113

Section 7.7: Commands to Control Buffers 115

Section 7.8: Commands for Working With Files..... 117

Chapter 8: The Cursor; Line Numbers; Point and Mark; The Region 121

Section 8.1: The Cursor and the Idea of Point 121

Section 8.2: Moving the Cursor 122

Section 8.3: Text Modes; Paragraphs and Sentences 123

Section 8.4: Repeating a Command: Prefix Arguments 124

Section: 8.5: Moving Through the Buffer 127

Section 8.6: Using Line Numbers 128

Section 8.7: Mark, Point, and the Region 129

Section 8.8: Using Mark and Point to Define the Region 131

Section 8.9: Operating on the Region 133

Chapter 9: Kill and Delete; Move and Copy; Correct Mistakes; Spelling; Fill 137

Section 9.1: Kill and Delete: Two Ways to Erase Text 137

Section 9.2: Commands to Delete Text 138

Section 9.3: Commands to Kill Text 141

Section 9.4: The Kill Ring and Yanking; Moving and Copying 143

Section 9.5: Correcting Common Typing Mistakes 146

Section 9.6: Correcting Spelling Mistakes 149

Section 9.7: Filling and Formatting Text 151

Chapter 10: Searching 155

Section 10.1: Introducing the Emacs Search Commands 155

Section 10.2: Incremental Searching 156

Section 10.3: Keys to Use While Searching 157

Section 10.4: Upper- and Lowercase Searching 160

Section 10.5: Non-Incremental Searching 161

Section 10.6: Word Searching 162

Section 10.7: Searching for Regular Expressions 162
Section 10.8: Regular Expressions 164
Section 10.9: Fixing Emacs Key Conflicts 166
Section 10.10: Searching and Replacing 167
Section 10.11: Recursive Editing 170

Chapter 11: Modes; Customizing Using Your `.emacs` File 173

Section 11.1: Introducing Modes 173
Section 11.2: Major Modes 174
Section 11.3: Lists of Major Modes 175
Section 11.4: Minor Modes 178
Section 11.5: Setting Major and Minor Modes 180
Section 11.6: Read-Only Mode 181
Section 11.7: Learning About Modes 182
Section 11.8: Customizing With the `.emacs` File; Learning Lisp 185
Section 11.9: Using Your `.emacs` File to Set Default Modes 187

Chapter 12: Shell Commands; Help and Info; Programs and Games 189

Section 12.1: Entering Shell Commands 189
Section 12.2: Shell Buffers 191
Section 12.3: The Help Facility 192
Section 12.4: The Emacs Tutorial; Info and the Emacs Reference Manuals 194
Section 12.5: Built-In Programs 197
Section 12.6: Built-In Tools, Including Dired 198
Section 12.7: Games and Diversions 203
Section 12.8: Zippy the Pinhead Talks to the Emacs Psychotherapist ... 212
Section 12.9: A Personal Note From Harley Hahn 214

Appendix A: Personal Notes 215
#1: Teaching Yourself Emacs 215
#2: Computer With a Keyboard 216
#3: Usenet, Emacs, and the Growth of the Internet 216
#4: Free/Open Source Software 217
#5: GNU's Not Unix? 217
#6: Our Tools Shape Our Minds 219
#7: AT&T 220
#8: Early Unix on the West Coast 220
#9: BSD Unix in the 1980s 220
#10: Hackers and Geeks 221
#11: Bash 221
#12: Linux Is Free 221
#13: Mac OS X Is Unix 222
#14: Terminals That Print 222
#15: Why U.C. San Diego in 1976? 223
#16: 80- and 132-character Lines 223
#17: Unix Workstations 224
#18: Time Travel 224
#19: Midnight Commander 225
#20: KDE and Gnome 225
#21: Aren't All Terminals Virtual? 226
#22: Ubuntu Terminal Emulators 227
#23: How to Access the Command Line With Mac OS X and Windows ... 227
#24: Freddy and the Men From Mars 228

#25: Special Files and Proc Files..... 228
#26: How Many Files Are on Your Unix System? 229
#27: Comparing Unix Packages to Commercial Apps..... 230

Appendix B: Command Summaries 231

Index of Emacs Key Sequences 257

Index of Emacs Variables and Functions..... 261

Index of Unix Keys, Files and Commands 263

General Index..... 265

About the Author

Harley Hahn is a writer, philosopher, humorist, abstract artist, musician and computer expert. Hahn has written 33 books, including three university-level Unix/Linux textbooks. In all, Hahn's books have sold more than 2 million copies, including *Harley Hahn's Internet Yellow Pages*, the first Internet book in history to sell more than 1 million copies.

Hahn is the best-selling Internet author of all time, and has had three of his books nominated for a Pulitzer Prize. His work—including a complete set of his books—is archived by the Special Collections Department of the library at the University of California at Santa Barbara.

In addition to books, Hahn has written numerous articles, essays, and stories on a wide variety of topics, including romance, philosophy, money and economics, culture, medicine, and biology. Much of his writing is available on his Web site `www.harley.com`.

Hahn has a degree in Mathematics and Computer Science from the University of Waterloo (Canada), and a graduate degree in Computer Science from the University of California at San Diego. He has also studied medicine at the University of Toronto Medical School, and has been the recipient of a number of honors and awards, including a prestigious National Research Council (Canada) post-graduate scholarship, and the 1974 George Forsythe Award from the ACM (Association of Computer Machinery).

Of all his endeavors, Hahn most enjoys writing books, because "I get to sleep in, and I like telling people what to do."

Web Site for This Book

For online support for *Harley Hahn's Emacs Field Guide*, please visit:

`http://www.harley.com/emacs`

At this Web site, you will find a variety of useful information. You can also send a message to Harley Hahn.

About the Technical Reviewer

Dmitry Shkatov has an MA in Philosophy from Moscow State University, Russia, and a PhD in Computer Science from the University of Nottingham, England. Shkatov lives in South Africa, where he is a Senior Lecturer in Computer Science at the University of the Witwatersrand, Johannesburg.

Acknowledgments

Writing a book requires me to spend many, many hours at home for months at a time, researching, composing and rewriting, alone at my desk, with only my dog Sadie to keep me company. (Sadie is a three-year-old border collie, who doesn't understand why we don't spend every day running around outside.) Publishing a book, however, is a team effort, so if you have a moment, I'd like to introduce you to my team.

Let's start with Lydia Hearn. Lydia and I have worked on books together for a long time, and I have found her to be a truly remarkable person. She is a tenured professor at De Anza College in Cupertino, California, where she teaches English. For this book, Lydia was my copy editor, which means that it was her job to find and correct any writing mistakes I might have made. This is harder than you might imagine, because writers think so fast that their fingers can't keep up with their thoughts, which means it is easy to make mistakes, many of which are subtle and hard to find. That is why I need Lydia, a tireless perfectionist (actually, a tired perfectionist) with boundless enthusiasm and enormous skill. Moreover, Lydia is the only person in the world that I will, voluntarily, let make changes to what I write.

Next comes Jeff Pepper, the lead editor for this book. Jeff and I started publishing books together a long, long time ago, and it is through his efforts that I have been able to create many of my most successful, high-quality books. Whenever I start a new project with Jeff, I know that he will be there with lots of advice, experience, and a good-natured approach that comes from dealing with authors and publishing people all day long, for many years. In addition, for this particular book, Jeff put in many hours—often late at night and under pressure — processing the files I would send him to ensure that everything was done just right.

The other editor on our team is Mark Powers, who served as managing editor for this book. This means that Mark handled a lot of details, including a schedule that seemed to have a life of its own. Mark has a background, not only in book publishing, but in writing graphic novels and comic books, which makes him uniquely qualified to work with the more esoteric parts of this book.

Then there is Dmitry Shkatov, a very special person whose job it was to help me look for technical mistakes. Dmitry teaches computer science at the University of the Witwatersrand in Johannesburg, South Africa. In fact, if it weren't for Dmitry, this book would never have been written. Here is how it happened.

On July 20, 2015, Dmitry wrote me the following letter:

Dear Harley,

I was wondering whether it would be possible to somehow get access to the chapter on Emacs from the 2nd edition of your "Student Guide to Unix", which was sadly left out of the 3rd edition.

It's the best introduction to Emacs I have ever seen (and that's how I came to love Emacs), and now that I'm teaching computer science to South African undergraduates, I'd like to be able to share it with my students.

Your assistance would be highly appreciated not only by me, but I am sure, by my students as well.

Not long afterwards, Jeff Pepper contacted me about writing a new book. I showed him Dmitry's letter, and we agreed that it would be a good idea for me to write a book focusing only on Emacs. We asked Dmitry if he would be the technical editor for the book, and he said yes. This was a stroke of luck for me, because you would have to search long and hard to find someone as smart as Dmitry, who has as much experience with Emacs. Dmitry not only read everything I wrote, making important suggestions, he also took the time to discuss all manner of Emacs-related ideas, which helped me enormously.

Speaking of help, I thank Ron Lockwood-Childs and Stephanie Lockwood-Childs, highly accomplished engineers and programmers, who helped me with advice on how to set up a Linux system to run on a virtual machine, specifically for testing Emacs. In addition, I thank Jebi Koilpillai, for helping me understand and test the technical issues involved in installing Emacs under Mac OS X.

As a writer, I always hope for production people who share my desire to create the best possible book. As such, I wish to thank several hard-working, skillful employees of SPi Global, an international business service provider. These three people are Mercy Thomas (account manager), Dulcy Nirmala Chellappa (project manager), Parameswari Balasubramaniyan (page layout artist), and Samuel Devanand (production editor). Without the time-consuming, detailed work of people like Mercy, Dulcy, Parameswari, and Samuel, it would not be possible to produce high-quality books like this one. I am lucky to have had their help.

For the final acknowledgments, I call upon the wisdom of the ancient Roman poet Juvenal who once observed, if you are going to spend hours a day for months at a time writing an Emacs book, you had better make sure that you get a lot of high-quality exercise, or you are going to have a lot of trouble. (Actually, the way he put it was *Mens sana in corpore sano*, which means "A healthy mind in a healthy body".)

With this in mind, I thank Sandrine Rocher-Krul, my cardioboxing teacher, for helping me to stay fit, alert, and healthy. I have taken a lot of classes in my life — as a child, as an undergraduate, in grad school, and in medical school — and without a doubt, Sandrine is the best teacher I have ever had. She has a rare confluence of motivation, skill, and creativity, which she demonstrates daily. For the same reason, I am grateful to the other students with whom I have been taking cardioboxing for so long. In particular, I thank Miguel Trujillo and Abe Solis for their continued encouragement and their help.

A Personal Note from Harley Hahn

You are floating down a narrow, winding river. Much of the time you move slowly, drifting with the current. You have been drifting a long time. Although the current is steady, the water is warm and comfortable, and you don't even notice it anymore. It just is. Warm, comfortable, predictable.

You are floating peacefully. Still, every now and then, you find yourself wondering: Is something missing? Am I bored? Is that all there is?

One day, you notice that the river is about to bifurcate into two different streams, and you need to make a choice. Which way do you want to go? You look carefully, squinting your eyes against the bright sun, trying to notice the details you are used to taking for granted.

Straight ahead, the water is calm. Do nothing, and you will keep drifting, safely and predictably. And why not? After all, your journey is far from over, and everything seems to be working out just fine. Part of you wants nothing more than to lay back and relax.

To the left, however, the water looks unusual. The current is faster and, in the distance, you see some rapids, although nothing you can't handle. A memory comes back: Years ago, you used to love the rapids, bouncing up and down for a short time, then resting to catch your breath. You were living in the moment, over and over and over, just being. How long has it been since you had that feeling of really being alive? You can't remember.

You close your eyes and imagine the map of the river, the map you thought you had memorized. However, you can't recall a place where one river turns into two. You open your eyes, feeling puzzled, and you see that, soon, you must make a decision.

Here is a promise. If you decide to turn and travel down the unknown branch of the river away from the current, you will find me there, waiting for you, and we will explore together. In fact, I have a map that we will share. There is a lot ahead of us, and I can tell you now that what is just around the bend will take effort. However, I can also tell you it will change your life.

What to do? Should you push against the current and explore the unknown, unexpected branch of the river? Or should you keep drifting in a slow, safe, predictable direction? Even doing nothing is a decision.

My advice? Go to Chapter 1 of this book and read the first three sections.

Then make your choice.

Harley Hahn

CHAPTER 1

All About Emacs

Section 1.1: Getting Started Together

Emacs is a powerful, computer-based tool, created a long time ago by smart people *for* other smart people. At its core, Emacs is a "text editor", a program you use to work with files that contain plain text. (We'll talk more about this in Section 1.2.) However, the Emacs text editor is part of a larger, more complex working environment and, in fact, there are many people — especially programmers — who happily spend most of their day working within Emacs.

In this book, I will teach you the basics of using Emacs. Although I will not be able to explain everything there is to know about Emacs — that would take several books — I will show you most of what you need to know for straightforward day-to-day work. In addition, before we get into a lot of technical details, you and I will discuss the environment in which Emacs was created and how it affects you even today. My goal is to ensure that, as you learn and use Emacs, you have a deep understanding of what you are doing and why.

Before we start working together, I'd like to get acquainted by talking about what I assume about you, my reader, and what I assume about the software you will be using. I will then tell you what you need to know about me.

Generally speaking, here is what we can say about the type of people who like to use Emacs (and because you are reading this book, I am assuming that these three statements apply to you):

1. Emacs users are smart.
2. Emacs users like to use computers.
3. Emacs users enjoy teaching themselves how to use complicated, powerful tools.

In addition, there is one more assumption I would like to make:

4. You are able to learn on your own by reading and practicing.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_1

This is important because, out of the 7.28 billion other people in the world, there is nobody who is ever going to teach you Emacs in person. You will have to teach it to yourself by reading and practicing.[1]

■ **Note** It wasn't always this way. In the 1970s, programmers and researchers worked in rooms with shared terminals, and programming was a much more social activity. Today, no one is going to teach you how to use Emacs in person, so you will have to teach yourself. Of course, you do have this book, so you aren't completely alone.

As you read the book you will, of course, need access to some type of Emacs, so you can practice and use what I will be teaching you. It will help a lot if you already have some experience using a computer with a keyboard.[2] Don't worry if Emacs is not already bundled (included) with your system: it is easy to find and it is always free.

There are a number of different versions of Emacs but, by far, the most widely used is GNU Emacs (which we will discuss in Section 1.4), so that is the version of Emacs we will be discussing in this book. GNU Emacs will work with just about every version of the Unix operating system — including Linux (all types), FreeBSD, NetBSD, OpenBSD, Mac OS X, Solaris — as well as with Microsoft Windows.

Once you have Emacs up and running, it works the same on any computer, under any operating system. For example, suppose you learn how to use Emacs with Windows. You then decide to change to Linux. The operating system and the working environment will be a lot different — but Emacs will look and function exactly the same.

Nevertheless, there are two important topics that do differ from one operating system to another: user interfaces and installation procedures. So when we cover these topics, I will be sure to show you how the details differ depending on your operating system. (Remember, though, Emacs itself will work the same.) In particular, since so many people use Emacs with some type of Unix (especially Linux), we will spend some time talking about operating systems in general (Section 2.1), and Unix and Linux in particular (Section 2.2).

To conclude this section, let's talk for a moment about what I can promise you. First, you can assume that I understand what I am teaching you, and that I have personally tested all the examples in the book.

Second, you can assume that I am experienced enough to explain highly technical details in a way that will be easy for you to understand and fun to read. My intention is that you should find learning Emacs to be an interesting and useful experience.

---

[1] See Personal Note #1, "Teaching Yourself Emacs" (Appendix A). From time to time, as you read as you read this book, you will see notes like the one above. In addition, there are also *personal* notes, which I hope you will find interesting and useful. For your convenience — and so I don't distract you — all the personal notes are collected in Appendix A.

[2] See Personal Note #2, "Computer With a Keyboard" (Appendix A).

Third, you can assume that I wrote this book very carefully, to be both a teacher and a reference. Specifically, there are 102 short sections, and everything you read in every section is something I want you to know. The very best way to use this book is to let it be your teacher. Start with the very first page and read straight through, in order, until you have finished all 102 sections. As you read, do not hurry. It will take a while to get to the part of the book where we actually start using Emacs. However, once we get there, please take the time to practice by trying all the examples, especially the many different key combinations, on your own computer. Although this may not seem necessary, it is the fastest and best way to learn Emacs. Taking time to practice the examples makes it easy for your brain to make the changes necessary for you to think like an Emacs person. (This is not a metaphor: you will need to give your brain time to change on the cellular level in order to use Emacs well.)

However, if this plan does not suit your needs, remember that this book is also a reference. If you so choose, you can skip around the book reading whatever sections you want in whatever order you want. As you read, you will find many forward- and backward-references to other sections, so if you encounter something you don't understand, it will be easy for you to find the information you need quickly.

As you read through the book, you will, from time to time, see two types of notes. First, you will see short notes (like the one below), useful and interesting comments placed within the text. Second, in the footnotes, you will see references to Appendix A, which contains longer (and more distracting) "personal notes". You will also see a few short discussions called "What's in a Name?", in which I explain the meaning of a specific name. For example, in Section 2.2, you will learn where the name "Linux" came from, and how to pronounce "Linus", the name of the person who started the Linux Project.

Finally, whenever I define the meaning of a new term, I will write it in capital letters, for example: EMACS is a sophisticated text editor within a complex and powerful work environment.

Note Emacs is difficult to learn, but easy to use.

Section 1.2: Emacs Is a Text Editor

What is Emacs?

Before I answer that question, let me take a moment to review some basic terminology.

Broadly speaking, there are two types of data that you can work with using your keyboard, mouse, and screen: text and graphics.

TEXT refers to data that consists only of characters that you can type on a keyboard: letters, numbers, punctuation, spaces and tabs. The old term for TEXT, which you will often encounter, is ASCII TEXT. (The name refers to one of the original systems used to represent characters stored as computer data, ASCII: the "American Standard Code for Information Interchange".)

GRAPHICS refers to any type of data that can be displayed on a screen, often using colors: not only characters, but images (photos or drawings); shapes (rectangles, circles); lines, dots, and so on.

A FILE is a collection of data that has a name and is stored on a disk or other storage device. Thus, a TEXT FILE or an ASCII FILE is a file that contains only text. A TEXT EDITOR (or more simply, an EDITOR) is a program used to create and modify text files.

EMACS is a text editor.

Text files have many different uses, so Emacs can be used for many different purposes. For example, you can use Emacs to write a story or an article, to make a grocery list or take notes during a lecture, or to edit system configuration files on a Linux system. In fact, any time you need to manipulate textual data directly, you need a text editor and Emacs is a good choice. In addition, Emacs is also capable of displaying images in a limited way.

■ **Note** When you start Emacs within a graphics environment (see Section 2.5), it will display a logo, which is an image. When you start Emacs within a text-only environment, you see only characters.

From the beginning, however, the most important use of Emacs has been to write programs, including shell scripts, and that is still the case today. Indeed, many people consider Emacs to be the best programmer's editor ever created. (I'll explain why in Section 1.3.)

■ **Note** A shell, also called a command processor, is the program that interprets the commands you type. A shell script is a list of such commands stored in a file. Shell scripts are commonly used to automate procedures or to write relatively simple programs. With Linux, the most commonly used shell is called Bash (see Section 2.2).

What else is Emacs used for? That depends on the state of technology and what people need at the time. For example, from the mid 1970s through the 1990s, many people used Emacs to compose email messages and to participate in Usenet (the global system of thousands of discussion groups). Today, using Emacs in these ways is obsolete.

Why? Today, almost everyone sends and receives email using a Web based-service or a modern email client (I use Eudora), all of which have built-in text editors, making Emacs unnecessary for email. Similarly, for over 25 years, Usenet was used by a very large number of people around the world to post messages discussing every topic you can imagine, as well as to develop and share software. Many people

used Emacs to create their messages, and a related program (**gnus**) to participate in Usenet discussion groups. Eventually, however, both discussions and software distribution migrated to the Web, which has no need of an external text editor.[3]

So how is Emacs used today?

Aside from programming, general note-taking, and so on as we discussed, you will find that many people use Emacs to create and modify files containing MARKED-UP TEXT: character-based information embedded with instructions as to how that information should be presented. The two most common examples are LaTeX files for typesetting documents, especially technical documents and research papers; and HTML and CSS files for creating Web pages.

The basic idea I want you to understand is that Emacs has always worked so well that, for over four decades, it has been used by smart people everywhere to work on whatever tasks arise in their lives that require a well-designed, powerful text editor. This is why learning how to use Emacs is such a good use of your time.

Note No matter what happens to come along in the future, you will always have a use for Emacs (as will your grandchildren).

Section 1.3: Emacs Is a Working Environment

There is a story in the Bible that relates what happened when the Queen of Sheba met King Solomon.[4] People had told the Queen that Solomon and his household were pretty hot stuff, but she was skeptical. Her attitude was that words are cheap, so she had better go to Jerusalem with some expensive gifts (just in case) and check things out for herself. When she got there, however, she was astonished: Solomon and his accomplishments were even more impressive than she had been told. When she finally met the king, she told him in person about all the stories she had heard and then remarked: "I believed not the words until I came, and mine eyes had seen it: and, behold, the half was not told me."

Well, that is what it is going to be like for you as you get more and more experienced with Emacs. In Section 1.1, I introduced Emacs to you as a text editor. But, as the Queen of Sheba might put it, the half was not told unto you: Emacs is more, a lot more. Here is why.

The text editor part of Emacs runs inside a framework that is highly customizable and very powerful. In fact, I would venture to say that Emacs is unlike anything you have ever seen or used before in your life. If you want to take the time, you can customize Emacs to work just the way you want and you can integrate Emacs with the other programs you use. In this way, you can build yourself an all-encompassing world in which you can spend all of your working time (and many people do).

---

[3] See Personal Note #3, "Usenet, Emacs, and the Internet" (Appendix A).

[4] Old Testament, 1 Kings 10:7.

■ **Note** For many people, Emacs is a way of life, like religion or football.

When you use Emacs on a Unix system, such as Linux, you can use any Unix command you want without leaving the program. For example, you can manipulate files and directories, test and run programs, download files from the Internet, install new software, and much, much more. You can also start a new shell (that is, open a new command line) whenever you want.

■ **Note** You can do pretty much the same thing using Emacs under Windows. However, the Windows command line is not nearly as powerful as the Unix command line.

In addition, Emacs is designed to make it very easy for you to change its behavior to suit your needs. For example, you can tell Emacs to work in a way that is suitable for the specific programming language you are using: indentation, syntax highlighting, and so on. You do this by setting what is called a "major mode" (see Section 11.2). For example, you can set major modes for programming in Assembly Language (x86), C, C++, Fortran, Go, Lisp, Java, JavaScript, MATLAB, Perl, Python, Ruby, and many other languages. Moreover, Emacs stays up to date. Whenever a new programming language starts to become popular, the Emacs community will create a major mode for it.

You can also set major modes for other types of editing, such as working with LaTeX, HTML and CSS files; multimedia files; picture files (including ASCII art!); and various types of data (such as CSV, comma separated value, files). In addition, there are many other modes (including the "minor modes", which we will discuss in Section 11.4) that enable you to fine-tune exactly how Emacs works for you.

If you are a programmer, you may be familiar with an IDE, or integrated development environment. An IDE provides a number of important programming tools with a common interface, in a way that makes them easy to use together:

- Text editor: to create and modify source code
- Debugger: to help track down and fix problems
- Build automation tool: to create a large program out of many components
- Help facility: to display documentation and reference material

For large projects, you will also use:

- Version control: to keep track of different versions of a program

■ **Note** SOURCE CODE, or more simply, CODE, refers to a collection of computer instructions written in a human-readable programming language, usually as text.

There are many different IDEs and, in a lot of cases, using one of them will be the best way to go. This is because they are complete packages, ready to go as soon as you install them. All you have to do is find the one that works best with the type of project on which you are working. However, with Emacs, it is possible to create your own, custom IDE. In fact, once you have enough experience with Emacs and Unix, you can set up your personal environment so that it is easy to access tools like the ones I mentioned above without ever having to leave Emacs.

For example, for some programming languages — such as C, C++, Python, and Java — Emacs is tightly integrated with build tools (**`make`**), a debugger (**`gdb`**), version control tools (Git), documentation tools, and so on. Emacs is also tightly integrated with all the programming languages that are part of the GCC. As a result, many programmers are able to work all day within the Emacs environment, leaving only to go to the bathroom or walk the dog.

Note As we will discuss in Section 1.5, GNU Emacs is part of the GNU Project from the Free Software Foundation (FSF). The FSF also maintains a compiler system, called GNU COMPILER COLLECTION or GCC. GCC is a complex, powerful tool that can compile a variety of programming languages (C, C++, Objective-C, Fortran, Java, Ada, Go) and generate code for many different processor architectures.

The more experience you have, the better you can set up Emacs to work exactly as you want it to, and you can change it instantly as your needs dictate. For example, Emacs lets you create as many "buffers" (work areas) as you want, each of which can be set up for a different task. This makes it easy to switch back and forth seamlessly between completely different types of work, according to what you want to do from one moment to the next. (We will talk about buffers in Section 6.2 and Section 7.7.)

If you are so inclined, you can extend Emacs by using a programming language, called Emacs Lisp, which is integrated into Emacs. Using Emacs Lisp, you can create what are called MACROS to enhance Emacs to suit your needs. A MACRO is a tool that you can create and use to perform a specific function according to your personal needs. Macros can be used for just about anything you can imagine, from simple abbreviations to use when you are typing, to complex tools that are actual programs in their own right. One of the most common uses of macros is to automate tasks that would otherwise be time consuming or error prone.

Finally, because Emacs is free, open source software, anyone can look at and modify the source code (the actual programs that make up Emacs) and contribute their own changes and improvements. This means that if you are willing to learn Emacs Lisp, you not only can customize your work environment up the wazoo, you can change Emacs itself by creating new tools and features, which you can then share with other users.

When I first introduced you to Emacs, I told you it is a text editor, which is true. However, we can now expand upon this definition: EMACS is a sophisticated text editor within a complex and powerful work environment.

There is a lot more I could say but we need to keep moving, so I'll end this section by quoting a short poem, called "Happy Thought", written by the Scottish poet Robert Louis Stevenson for the Queen of Sheba to explain to her the pleasure that comes from using Emacs:

The world is so full of a number of things,
I'm sure we should all be as happy as kings.[5]

■ **Note** If you want to master Emacs, it helps to believe in reincarnation, because there is no way you are going to learn it all in a single lifetime.

Section: 1.4: Where Did Emacs Come From?

Many people believe that Emacs is of divine origin, but that is only partially correct.

The original Emacs was developed by Richard Stallman at MIT in 1975. At the time, Stallman was working in the MIT Artificial Intelligence Lab on an operating system called ITS, the Incompatible Time-sharing System, using a PDP-10 computer. (The name was coined to make fun of CTSS, the Compatible Time-sharing System, a more mainstream system that had been developed at the MIT Computation Center.)

One of the programs used by many ITS users was a text editor named TECO. Although TECO was useful, it was complex and difficult to use, driving even experienced programmers to the point of dementia. To make their lives simpler, some of the programmers developed collections of macros whose purpose was to make TECO easier to use. (As we discussed in Section 1.3, a macro is a tool that you can use to perform a specific function according to your personal needs.) In 1976, Stallman organized and extended the various sets of TECO macros into a coherent collection, which collectively, were referred to as the Emacs text editor. (The name is explained below.)

Since then, Emacs has been rewritten as a separate program many times, and greatly improved. It is available in a number of versions, the most popular being GNU Emacs, a product of the Free Software Foundation (FSF), which we discuss in Section 1.5. This is the version of Emacs you are most likely to encounter on a Unix system. Most of GNU Emacs is written in Emacs Lisp; the rest is written in C.

---

[5] Robert Louis Stevenson. *A Child's Garden of Verses*, 1885.

What's in a Name?

TECO, Emacs

The original version of Emacs was a set of editing macros written to run under the TECO editor. Originally, the name TECO stood for "Tape Editor and Corrector". Later, the official name was changed to "Text Editor and Corrector".

Emacs is a simple abbreviation for "Editing Macros".

Section 1.5: The Free Software Foundation

In 1985, ten years after he released the first version of Emacs, Richard Stallman founded the FREE SOFTWARE FOUNDATION or FSF, based on the belief that high-quality software should be readily available without the usual commercial restrictions. Toward this end, Stallman wrote a manifesto (see Section 1.6) in which he set forth his philosophy regarding "free software". Stallman's influential and enduring contributions to programming culture were based on an observation:

- Programmers, by their nature, like to share their work.

From which he formed two core beliefs:

- When programmers have such freedom, the world benefits.
- Software licensing restrictions should reflect this reality.

It is important to understand that Stallman did not mean that all software should be available for no cost, with no restrictions. Rather, in calling for "free" software, Stallman was referring to freedom, not price. (At the time, it was popular to say that Stallman used the word "free" as in "free speech", not "free beer".) By definition, FREE SOFTWARE refers to programs that are distributed with a license specifying that anyone in the world is allowed to read the source code, modify the code, and share the results of their work freely. Moreover, any program based on free software must itself be licensed as free software. Another name commonly used for free software is OPEN SOURCE SOFTWARE.[6] (Programs that are not free of such restrictions are called PROPRIETARY SOFTWARE.)

To describe this philosophy, the FSF created the GNU GENERAL PUBLIC LICENSE or GPL, which mandates that any program derived from free software is itself free software. This is why any version of Emacs you will ever find is free software: all versions of Emacs are derived, at least indirectly, from GNU Emacs, which is distributed using the GPL.

However, in starting the Free Software Foundation, Stallman had more in mind than being able to distribute Emacs as free software. His goal was much more ambitious: to create a complete, free version of a Unix-like operating system, which he called GNU, to be licensed under the GNU General Public License.

[6] See Personal Note #4, "Free/Open Source Software" (Appendix A).

■ **What's in a Name?**

GNU

GNU is the name Richard Stallman chose to describe the Free Software Foundation's project to develop Unix-like tools and programs. The name GNU is an acronym for GNU's Not Unix.[7]

GNU is pronounced "ga-new", to rhyme with the sound that you make when you sneeze.

Section 1.6: Excerpts From *The Gnu Manifesto*

As I mentioned, Richard Stallman (Figure 1-1) wrote a manifesto whose philosophy forms the foundation of the Free Software Foundation. His basic idea — that *all* software should be shared freely — is, at best, naive. However, with the rise of the Internet, the development and distribution of free software and open source software has become an important economic and social force in our world. There are literally tens of thousands of programs available for free, and their contribution to the world at large (and to the happiness of their programmers) is beyond measure.

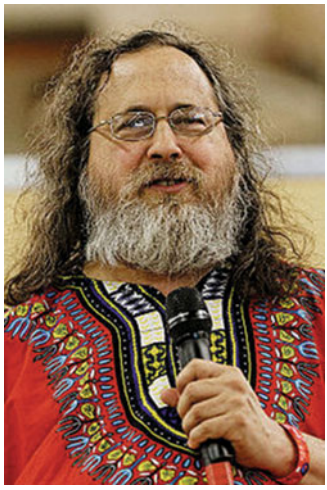

FIGURE 1-1. Richard Stallman, 2014.

Since 1975, Richard Stallman (1953-), the creator of Emacs and founder of the Free Software Foundation, has been a tireless and influential advocate for the importance of free software.

The FSF has been one of the leaders in this area, not only with Emacs, but with a C compiler (**`gcc`**), a C++ compiler (**`g++`**), a powerful debugger (**`gdb`**), a Unix shell (**`bash`**), and many, many other tools. All of this software — which is part of the GNU project — is used around the world and is considered to be of the highest quality.

Stallman's public declaration was not as sophisticated as other well-known manifestos, such as *95 Theses* (Martin Luther, 1517), or *The Playboy Philosophy* (Hugh Hefner, 1962-1966). Still, the work of the Free Software Foundation continues to make an important contribution to our culture and, for this reason, you may be interested in reading a few excerpts from Stallman's 1985 essay.

If you want to read the entire *GNU Manifesto* (it's not long), it is easy to find online: just search for "gnu manifesto". You can also read it within Emacs by displaying the "Manifesto" section of the online Emacs manual. Here is how to do it. (These instructions will make sense once you learn some basic Emacs commands.)

[7] See Personal Note #5, "GNU's Not Unix?" (Appendix A).

1. Start Emacs
2. Start the Info facility: type <Ctrl-H> **i**
3. Jump to the menu item named "Emacs": type **m** then **emacs**
4. Jump to the menu item named "Manifesto": type **m** then **manifesto**
5. To move around as you read: press <PageUp> and <PageDown>
6. To quit: press <Ctrl-X> <Ctrl-C>

In addition to *The GNU Manifesto*, there is another important essay, called *The GNU Project*, also written by Richard Stallman,that will give you the flavor of the thinking behind Emacs and the Free Software Foundation. If you are interested in the philosophy behind free software, you can find this essay online by searching for "about the gnu project". You can also read it from within Emacs, as follows:

1. Start Emacs
2. Start the "About Emacs" part of the Help facility: type <Ctrl-H> then **g**
3. To move around as you read: press <PageUp> and <PageDown>
4. To quit: press <Ctrl-X> <Ctrl-C>

To finish this section, I have included, below, a few passages from the original GNU Manifesto. Note that when Stallman says software should be free he does *not* mean that anyone — including for-profit corporations — should be able to use any program for no money. He means that no one should have to pay for *permission* to use a program, although there may be a charge for distribution or support.

The following are excerpts from *The GNU Manifesto*:

"I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will. So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away…"

"Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money…"

"Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free…"

"In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming…"

CHAPTER 2

Unix for Emacs Users

Section 2.1: Operating Systems

I have already mentioned Unix several times. Unix is the name of a family of operating systems, and it is important to us for three reasons. First, most people who use Emacs do so with a Unix system such as Linux (which is a type of Unix). Second, Emacs was developed — and still lives — within the Unix culture. Finally, operating systems are important, and Unix and Linux are interesting in their own right. So before we get into the details of using Emacs, I'd like to take some time to talk about operating systems generally, and Unix and Linux in particular. I'm going to start with a few basic ideas, so if you are already an experienced programmer, please be patient. Although you might already be familiar with this material, I would ask you to at least skim it, just to make sure.

Whenever a human being uses a "computer system", there are three fundamental parts of the system that interact with one another: the computer itself, the programs that run on the computer, and the person who uses the programs. These three parts are inter-dependent:

- Computers (machines) run programs in order to perform tasks for people and provide services to other programs.
- Programs bring life to what would otherwise be inert machines.
- People use computers for a large variety of purposes, including writing more programs to run on computers.

When we talk about these ideas, we generally use more technical, impersonal terms. Specifically, we call the machines HARDWARE; we call the programs SOFTWARE; and we call the people USERS. The important idea I want you to appreciate is that computer systems are complex entities, created by a melding of hardware, software, and users. Since you are the user, you are affected by the hardware and the software that you choose to use. And, whether you realize it or not, one of the most important choices you make in your life is which operating system you will be using.[1] So let's talk about that for a moment.

[1] See Personal Note #6, "Our Tools Shape Our Minds" (Appendix A).

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_2

An OPERATING SYSTEM is a complex master control program that makes a computer work. Every computer, from the smallest phone to the largest supercomputer uses an operating system. Traditionally, the most important jobs for an operating system are:

1. To make efficient use of the hardware.
2. To furnish an environment for software, offering resources to the various programs as they run.
3. To provide an interface for users to interact with the software.

Since most operating systems are bundled (packaged) with a large number of tools, we can add one more item to our list:

4. To offer a variety of basic tools (useful programs) for users.

There are a large number of different operating systems. However, the ones that you and I are likely to encounter are almost always from one of two very different families: Unix and Windows.

WINDOWS, more formally, MICROSOFT WINDOWS, is the name of a family of commercial operating systems, first released in 1985. Since then, Microsoft has created many versions of Windows to run on a large variety of different types of hardware. Windows is a commercial product, created and controlled by Microsoft. If you want to use Windows, you (or someone) has to pay for it. UNIX is a general term, referring to a large number of operating systems that are based on a large set of specific principles and specifications developed over many years. Unix is not owned by a single company. Moreover, there are many versions of Unix that are available for free.

The important thing to appreciate is that, as long as you are using a computer that has a keyboard (not a tablet, phone, or touchpad), Emacs is available — for free — to run on whatever system you are likely to be using. And (you will have to believe me here) it will be good for your brain to learn how to use Emacs.

However, before we get into the details of installing and using Emacs, I want to take some time for us to talk about Unix.

Section 2.2: Unix and Linux

Unix is the name of the world's largest family of operating systems.

The first primitive version of Unix was created in 1969 by Ken Thompson, a programmer at the Bell Labs research facility in New Jersey, owned by AT&T.[2] Within a short time, Unix became popular among the Bell Labs' and other East Coast researchers. The two main Unix pioneers at Bell Labs were Ken Thompson and Dennis Ritchie. (Their names are worth remembering.) Throughout the 1970s and into the early 1980s, Thompson, Ritchie, and other programmers expended a great deal of effort expanding and improving Unix.

[2] See Personal Note #7, "AT&T" (Appendix A).

In 1974, the use of Unix began to spread to the West Coast, when a computer science professor from the University of California at Berkeley visited Bell Labs and brought back a copy of Unix. In 1975, Ken Thompson went to Berkeley from Bell Labs for a year-long sabbatical. As a result of Thompson's visit, a graduate student named Bill Joy became interested in Unix and began to work on his own version. The result was a new Unix, called BSD (an acronym for Berkeley System Distribution), which became popular among computer scientists and programmers around the world.[3] In fact, today, there are still a variety of Unix systems based on BSD, the most important being FreeBSD, NetBSD, OpenBSD, and DragonFly BSD. In addition, Apple's Mac OS X operating system (often referred to as OS X) is, to a great extent, also based on BSD. (See the footnote in Section 2.3.)

By the early 1980s, AT&T — the company that owned Bell Labs — became increasingly impatient to start making money from Unix. To do so, in 1983, AT&T turned Unix into a commercial product called System V (pronounced "system five"). Throughout the rest of the 1980s, a large number of different Unix systems were developed, based on either System V (what we might call East Coast Unix) or BSD (West Coast Unix). The System V-based versions of Unix were commercial products, developed and sold by computer companies, including AT&T. Most of the BSD-based versions of Unix, on the other hand, were non-commercial and were distributed for free.[4] In this way, BSD gathered a large following of computer scientists, researchers, and programmers around the world.

By 1991, personal computers had been around for 10 years, and many programmers now had their very own computers. (The IBM Personal Computer was introduced in August 1981.) However, there was still no operating system that ran on a PC that was attractive to the type of programmer whose idea of fun was to take things apart and modify them.[5] Nevertheless, the world was ready. What follows is a long, interesting story, very much related to Emacs. However, in the interest of brevity I will shorten it.

In 1985, Richard Stallman started the Free Software Foundation (FSF). As we discussed in Section 1.4, one of Stallman's goals was to create a free version of Unix — which he named GNU — that would be available to anyone who wanted to modify, use, or distribute it in any way. Nevertheless, by 1991, GNU was not nearly ready. To explain what happened next, I have to make a quick technical digression. However, it's an important digression, so stay with me.

As we have discussed, operating systems such as Unix are very complex. To simplify a bit, the *basic* functionality of a Unix system is provided by two different parts: the kernel and the utilities. The KERNEL, which is always running, is the central part of the operating system. As such, it provides essential services as they are needed. The UTILITIES consist of a wide variety of separate programs, distributed

---

[3] See Personal Note #8, "Early Unix on the West Coast" (Appendix A).
[4] See Personal Note #9, "BSD Unix in the 1980s" (Appendix A).
[5] See Personal Note #10, "Hackers and Geeks" (Appendix A).

along with the kernel as part of the entire system. Where the kernel is the heart of an operating system, the utilities provide functionality to the users and to other programs. To summarize:

Kernel + Utilities → basic operating system

Unix Kernel + Unix Utilities → basic Unix system

Modern Unix systems come with, literally, hundreds of different utilities, each of which provides a different function. The most important utility is the SHELL — also known as the COMMAND PROCESSOR — the program that provides the primary user interface into Unix. The job of the shell is to process Unix commands as they are entered by a user. The shell can also process a list of commands stored in a file, called a SHELL SCRIPT. Over the years, there have been a variety of different Unix shells. Today, the most popular shell — the one you will probably use — is called BASH.[6] (In Section 2.7, we'll talk about how to enter commands for the shell.)

To return to our discussion: By 1991, the world was ready for a completely free operating system that programmers could run on their own PCs for fun, for experimenting, and to modify and share — without cost — with other programmers around the world. (The Microsoft and Apple operating systems, Windows and Mac OS respectively, didn't fill this need because they were — and still are — purely commercial products that are kept under tight control by the companies that own them.) To be sure, the FSF had done a lot of work towards creating an open version of Unix for personal computers. By 1991, they had created a great deal of high-quality, free software, including many of the traditional Unix utilities (and Emacs). However, the FSF did not, as yet, have a kernel. And until such a kernel was created, there would be no free Unix system for PCs.

However, on August 25, 1991, Linus Torvalds (Figure 2-1), a second-year computer science student at the University of Helsinki (Finland) announced via Usenet,[7] that he was developing his own freely shared Unix-like kernel for PCs. Torvalds started this project just for fun, and asked for volunteers. He realized that all he really needed was a kernel, because the FSF had already created a free version of the most important Unix utilities, which he could adapt easily to his own system.

In September 1991, Torvalds and his collaborators released the first-ever free Unix kernel to run on a PC. It was then a relatively easy job to adapt the FSF utilities to work with the new kernel, giving birth to a brand new version of Unix, which came to be known as LINUX. Linux is free,[8] open source software, licensed primarily under Version 2 of the GNU General Public License (GPLv2). (We discussed the GPL in Section 1.5.)

---

[6] See Personal Note #11, "Bash" (Appendix A).

[7] The worldwide system of discussions groups. See Personal Note #3, "Usenet, Emacs, and the Internet" (Appendix A).

FIGURE 2-1. Linus Torvalds, 2004.

Linus Torvalds (1969), the original creator of the Linux kernel. In 1991, Torvalds founded the project that resulted in the first-ever, free Unix kernel. Torvalds combined his kernel with the Free Software Foundation's utilities to create Linux, the first free Unix system for personal computers. Today, there are a very large number of different Linux systems, running on virtually every type of computer.

Today, there are many, many different versions of Linux — often referred to as Linux DISTRIBUTIONS — running on virtually every type of computer: from tiny computers that are smaller than your hand, to the largest supercomputers.

Note If you would like to try Linux and you are not sure which distribution to choose, my advice is to start with Ubuntu Linux.

---

[8] See Personal Note #12, "Linux is Free" (Appendix A).

■ **What's in a Name?**

Linux

When Linus Torvalds was working on his first kernel, he sometimes referred to it informally as Linux, the name being a contraction of "Linus' Minix". (Minix was a small, Unix-like operating system created by the Dutch computer scientist Andrew Tannenbaum.) However, when it came time to release the new kernel, Torvalds had decided to name it Freax, for "free Unix". Here is where fate steps in.

In September 1991, it happened that another programmer, Ari Lemmke, convinced Linus to distribute the kernel files using an FTP (file sharing) server maintained by Funet, a Finnish academic/research network. However, Lemmke didn't like the name Freax and when he received the files from Torvalds, he uploaded them to a directory he called **`linux`**, and the name stuck.

Today, the name Linux refers to any operating system that uses the Linux kernel. Many Linux systems still use the FSF utilities and, for this reason, you will sometimes see Linux referred to as GNU/Linux.

The name Linux is pronounced to rhyme with "bin'-ex".

The name Linus is pronounced "Lee'-nus".

Section 2.3: Unix Terminals and Userids

In Section 1.4, I told you that the most popular type of Emacs is GNU Emacs, and that it runs under a variety of different operating systems, including Microsoft Windows. However, almost everyone who uses Emacs does so under some type of Unix. Specifically, GNU Emacs is available in specific versions for Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X,[9] and Solaris. For this reason, I am going to assume that there is a good chance that you, too, will be using Emacs on a Unix system. To prepare you properly, there are important Unix concepts I want to make sure you understand.

What we are about to cover in the rest of this chapter is particularly relevant to learning how to use Emacs skillfully. However, it is, necessarily, a brief, abbreviated overview of Unix. There is a *lot* more to learn and, unfortunately, I am not able to cover it all in this book. The best way to learn how to use Unix/Linux well is to read my book *Harley Hahn's Guide to Unix and Linux*.[10] As you read this chapter, you might wonder, is all this really necessary? The answer is, yes. I can assure you that people who are skilled at using Emacs know everything I will be covering. Please take the time to read through all the material.

[9] See Personal Note #13, "Mac OS X Is Unix" (Appendix A).

[10] *Harley Hahn's Guide to Unix and Linux*, McGraw-Hill Higher Education, 2008. The ISBN is 0073133612.

To start, I'll remind you that Unix was developed in the 1970s (see Section 2.2), before programmers were able to have their own personal computers. To work with Unix, a person would use a terminal to connect to a host. The TERMINAL was an electronic box with a keyboard and a screen, or a keyboard and a printer.[11] The HOST was the computer to which the terminal connected. The connection could be either via a cable (which was relatively fast), or via a modem and telephone line (which was very slow). Such terminal/host systems were called TIME-SHARING SYSTEMS or MULTIUSER SYSTEMS, because they could support multiple users who were "sharing" the computer's time, so to speak.

Thus, from the very beginning, the architecture of Unix was based on a terminal/host time-sharing system and — believe it or not — that is still the case today. I want to spend a moment talking about this paradigm because it will help you understand what happens when you use Unix in general and Emacs in particular.

Let's pretend for a moment that we are going to take a time-travel trip back to the late 1970s. Close your eyes and pretend that it is 1976, and you and I are visiting the computer science department at U.C. San Diego.[12] We want to use one of the Unix systems so we visit the SYSTEM ADMINISTRATOR, the person who runs the system we want to use. The system administrator sets up a Unix account that will enable us to use the system. The ACCOUNT includes a variety of information including a userid and a password.

A USERID (pronounced "user-eye-dee") is a one-word name, usually all lowercase, that is used to identify a user to the system. A PASSWORD is a (hopefully) difficult-to-guess series of characters that provides security for a particular userid. Passwords are secret; userids are not.

As an example, let us say that the system administrator gives us the userid **`harley`** and a password **`kajsaanka.`**[13] Whenever we want to use Unix, we must find a terminal and identify ourselves to the system. To do so, we LOG IN by typing our userid (so the system knows which account we will be using), followed by our password (to prove that we are allowed to use that account).

After logging in, we can now make use of the Unix system by entering commands. As we discussed in Section 2.2, the program that interprets our commands is called the shell. Each time we log in, a shell is started for us automatically. It then sits there quietly, waiting to serve us. We enter commands, one after another, and when we are finished using the system, we LOG OUT by terminating the shell. We do this by entering the **`logout`** command, or by pressing a special key combination (<Ctrl-D>) to tell the shell there are no more commands.[14] Once we have logged out, someone else can use the terminal.

---

[11] See Personal Note #14, "Terminals That Print" (Appendix A).
[12] See Personal Note #15, "Why U.C. San Diego in 1976?" (Appendix A).
[13] Kajsa Anka is a real name. Look it up.
[14] Pressing <Ctrl-D> sends an **`eof`** signal, which indicates that there is no more data (see Section 2-6).

Aside from keeping track of our userid and password, our account also specifies what level of privileges we are allowed to use. Generally speaking, there are two types of userids: the system administrator's and everyone else's. The system administrator has a special userid with its own password that allows him to do *anything he wants*. When a person logs in using this userid, we say that he or she becomes SUPERUSER. To protect the integrity of the system, all the other accounts have restricted privileges: they are able to use the system for regular work, but they do not have enough power to cause harm to the system or to other people's files. Being superuser is so powerful that it must be used sparingly to protect the integrity of the system and to keep everyone's files safe.

■ **Note** Traditionally, the name of the system administrator userid is **`root`**. The name was chosen because the main directory in the Unix filesystem is called the root directory (see Section 2.16).

Typically, a system administrator will use a regular account for his own personal work. He or she will only use the superuser account when absolutely necessary. In our example, when we asked the system administrator for a Unix account, he would have paused what he was doing, logged in as superuser, created a new account for us, and then logged out as superuser.

So that is how it worked in 1976. To use a Unix system, we had to ask the system administrator to create an account for us, with a userid and password. We then had to find a terminal attached to that system, and log in. We would use the system by typing commands, one after the other, which would be processed by the shell. When we were finished, we would log out. If something important had to be done that required special privileges, the system administrator was required to log in as superuser, at least long enough to carry out that particular task.

The reason I am telling you all this is because — believe it or not — decades later, Unix still works the same way. And since Emacs developed within a Unix environment, the Unix way of thinking is deeply connected to the Emacs way of thinking.

Consider the following metaphor. Let's say you want to learn how to speak a new language. Ideally, you want more than to simply speak the language; you want to speak it without an accent. It's the same with Emacs. No matter what operating system you use to run Emacs; no matter how much time you spend with it; no matter what you use it for — if you don't learn basic Unix, you will never be able to speak Emacs without an accent.

Eventually, this will all make sense. For now, just trust my judgment and keep reading as we move towards our next goal: to discuss the various environments ("user interfaces") that you can use to run Emacs. However, for that discussion to make sense, we first need to take a few minutes to talk some more about terminals.

Section 2.4: Types of Terminals

The very first computer terminals, dating from the 1960s, were PRINTING TERMINALS that printed their output on paper: either a continuous roll or a stack of folded, perforated pages (called COMPUTER PAPER). You can see two such terminals in the photo in Figure 2-2. This photo, taken around 1970, shows Dennis Ritchie and Ken Thompson working on a PDP-11 computer at Bell Labs. It is likely that this is the very computer used by Ritchie and Thompson to create the first version of Unix.

FIGURE 2-2. Dennis Ritchie (standing) and Ken Thompson (sitting), circa 1970.

This photo, taken at Bell Labs sometime around 1970, shows Dennis Ritchie (1941-2011) and Ken Thompson (1943-). Ritchie and Thompson, the original developers of Unix, are working on a DEC PDP–11 computer. The terminals that Thompson is using are Teletype Model 33 ASRs.

By the early 1970s, printing terminals were being replaced by VIDEO DISPLAY TERMINALS. Instead of using paper, video display terminals displayed their output on a built-in monitor. The terminal worked with a set number of rows, and each new line of text was written to the bottom row of the monitor. As this happened, all the other lines were moved up one row, while the (old) top line disappeared.

The DEC VT52 terminal (introduced in July 1974) had 24 rows, each of which could display 80 characters. A few years later, the VT52 was replaced by a more powerful terminal, the DEC VT100 (August 1978), which displayed either 24 rows of 80 characters or 14 rows of 132 characters.[15] Since this type of terminal can display only characters, we refer to them as TEXT TERMINALS. You can see a photo of a VT100 in Figure 2-3.

As I mentioned, the original video display terminals displayed new data only on the bottom row, causing all the lines to move up (similar to a printing terminal). However, as text terminals were improved, it became possible for the software to display or erase characters in any position on any row. This meant that programmers now had more choices. Instead of simply displaying output on the bottom line — causing all the other lines to move up — terminals such as the DEC VT100 made it possible for programs to use the entire screen at once, typically 1,920 separate characters (24 lines x 80 characters/line). This enabled programmers to create a new type of interface. We'll talk about the details in Section 2.5. For now, I'll just say that this is the type of interface that was used to create Emacs.

FIGURE 2-3. VT100 terminal.

The DEC VT100 terminal was introduced in 1978 by the Digital Equipment Company. In the late 1970s and early 1980s, these VT100 family of terminals were very popular. To this day, their basic characteristics are preserved in most terminal emulators.

Over the years, better and better video display terminals were developed: not only text terminals but, eventually, GRAPHICS TERMINALS that could display both text and images. A few of the more widely used graphics terminals were the IBM 2250 (introduced in 1964), the DEC GT40 (1972), the IBM 3270 (1977), the Textronix 4010 family (1972-1974), and the latecomer DEC VT240 (1984).

Graphics terminals were important because they made it possible for programs to display images, lines, curves and other shapes, as well as plain text. Graphics terminals also made it possible to share graphical programs over a network, a capability that was especially important to universities and research organizations. However, compared to text terminals, graphics terminals were very expensive and required much more complicated software to connect to a host computer. One

[15] See Personal Note #16, "80- and 132-character Lines" (Appendix A).

reason was that every type of graphics terminal had its own distinctive requirements, and it was a lot of work to make even a simple program work with different types of graphics terminals. This is because displaying graphical output requires special-purpose hardware and extra computing power.

In the late 1980s, however, these limitations began to change because of an ambitious and, ultimately, very successful project called X Window, started at MIT in 1984 by Bob Scheifler and Jim Gettys. At the time, there were many MIT programmers, using a large variety of different types of hardware. What X WINDOW offered was a standard way for people to use the keyboard, monitor and mouse on their own computer to interact with programs that were running on any computer in their network. Moreover, X Window offered portability. Once a program was written to work with X Window, it could be used by anyone whose computer had X Window software.

Because personal computers with graphics capabilities were now readily available, it wasn't long before computers running X Window obviated the need for expensive, complex graphics terminals. Within a short time, X Window became an industry standard that, over the next few years, led to a revolution that would change forever the way most people used their computers.

What fueled this revolution was the growing realization that it was not the case that people interact with hardware. Rather, people *use* hardware to interact with programs. Although this might seem obvious to you now, this idea was an important insight that took years for people to appreciate. Why this was so important and, specifically, how it influenced the development of Emacs, is the topic of Section 2.5.

Section 2.5: User Interfaces

On August 12, 1981, at the Waldorf Astoria ballroom in New York City, an event occurred that would, in retrospect, be recognized as changing the world. At an IBM press conference, an executive named Phil Estridge announced the brand new IBM 5150 computer, nicknamed the IBM Personal Computer or IBM PC. Since the late 1970s, other companies had been making relatively inexpensive computers to be used exclusively by one person at a time. The most popular machines were the Commodore PT, Apple II, TRS-80 and Atari 400. However, IBM was, by far, the largest computer company in the world, and it wasn't until they finally announced their own personal computer, along with a new set of industry standards, that the age of personal computing finally began.

Note The cost of the original IBM PC was $1,565 (a bit more than $4,000 in 2016 dollars). The original IBM PC came with 16 kilobytes of memory (0.000016 gigabytes) and no hard disk. Instead it had a disk drive that used floppy disks which could store up to 160 kilobytes of data.

By the end of the 1980s, personal computers became more affordable and more powerful, and Unix programmers had shifted from using terminals to using their own computers. Specifically, many people were using Unix on inexpensive IBM PC COMPATIBLE personal computers, machines that adhered to standards based on the IBM PC family. Other people, who needed more power and larger monitors, used expensive, high-end Unix machines called WORKSTATIONS.[16]

Remember, however, Unix had always been a multiuser, terminal/host time-sharing system (and it still is). As we discussed in Section 2.3, in the early 1980s, to use Unix you needed a terminal to access a Unix host. But with a personal computer, all the hardware is part of a single machine. So the question arises: When you use Unix on your own computer, where is the terminal and where is the host?

The answer to this question involves two important inventions: "terminal emulators" to use with text-based programs and "graphical user interfaces" to use with graphics-based programs. We'll start with the text-based programs.

When you run a text-based program on your own Unix computer, even today, you need a terminal to connect to a host. That's the way Unix was designed in the early 1970s, and it has never been changed. As you might guess, your computer is the host. But where is the terminal?

Obviously, you don't use a real terminal. Instead, you use a special program that utilizes your computer's keyboard and monitor to simulate the functionality of a terminal. In this way, you can log in, enter as many commands as you want, run as many programs as you want, and log out, just as if you were using an actual physical terminal.

When a program simulates the functionally of a physical device, we say that it EMULATES that device. So a program that acts as a terminal is called a TERMINAL EMULATOR. There are many different terminal emulators in use today and, believe it or not, they are all based on the old DEC VT100 family of terminals we discussed in Section 2.4. This means that, even today, when you type a Unix command or run a text-based Unix program, you are using a technology that was first introduced in 1978.[17] Moreover, as far as Unix is concerned, the terminal/host system is working just fine, the way it has always worked, because your programs don't know whether you are using a terminal emulator or a real terminal. This concept is important because the underlying technology has a strong influence on how you interact with your programs. Let me be more specific.

To communicate with a program that is running, we use what is called a USER INTERFACE. There are two basic text-based interfaces, which means that there are two different ways in which you might find yourself working with text-based programs. (Graphics-based programs work differently, as you will see in a moment.)

---

[16] See Personal Note #17, "Unix Workstations" (Appendix A).
[17] See Personal Note #18, "Time Travel" (Appendix A).

The simplest text-based interface is the COMMAND-LINE INTERFACE or CLI. When you use a CLI, you type one line at a time and then press the <Enter> (or <Return>) key. As you type, the characters are displayed on the bottom line of the screen. When you press <Enter>, whatever you have typed is sent to the program, and all the lines on the screen scroll up one row. In the same way, when a CLI-based program writes output for you to read, it does so one line at a time on the bottom line of your screen. As each line is displayed, all the other lines scroll up one row.

When you use a CLI, you use only a keyboard for input, not a pointing device. The most common example would be typing Unix commands, one at a time, for the shell to interpret. (This is where the name "command-line interface" comes from. The bottom line of the screen where your input is displayed is called the COMMAND LINE.)

The second text-based interface is more sophisticated. It is called a TEXT-BASED USER INTERFACE or TUI. With a TUI, a program can read and write characters anywhere on the screen, not just on the command line. As a result, TUI-based programs have a different look and feel: they take over the entire screen, so you have the feeling that you are working "within" the program. The more sophisticated TUI programs may display colors, and will use the various parts of the screen in different ways. For example, a TUI-based program may arrange data into columns, or even create rudimentary windows.

Some TUI programs will let you use a mouse, most commonly to access simple menus to make selections. Still, with a TUI-based program, pull-down menus are generally not very important. Most of the time, you will be using your keyboard to control the program, which means that you will need to memorize a lot of different commands. The reason I am telling you all this is because Emacs is a TUI-based program. In fact, it is one of the most powerful TUI-based programs ever written.[18]

When you work with graphics-based programs, you use a GRAPHICAL USER INTERFACE or GUI, which is a lot different than a CLI or TUI. A GUI manages the entire screen, or multiple screens if you have more than one monitor. With a GUI, you typically run multiple programs at the same time, each program in its own window. Whenever you change to another program — usually by clicking on its window — we say that you move the FOCUS to that program. (At any moment, the program that has the focus is the one that reads the input from your keyboard.)

When you use a GUI, you will see all the typical graphical elements: windows, icons, menus, scroll bars, buttons, and so on. The background on which these elements appear is called the DESKTOP. (For example, you might say, "I have three windows open on my desktop.") For this reason, a GUI-based working environment is often referred to as a DESKTOP ENVIRONMENT. More specifically, a desktop environment refers to the design of all the graphical elements, how they work, and

[18] See Personal Note #19, "Midnight Commander" (Appendix A).

how they interact; as well as all the tools that are included as part of the GUI package, such as a file manager, a terminal emulator, tools to help you change your system's settings, and much more.

When you install Linux, you create a master account by specifying a userid and password. Each time you start Linux, you will be asked to enter your userid and password, at which point the GUI will start automatically. In the world of Unix, there are many GUIs, but they are all based on the X Window system we discussed in Section 2.4. With Linux, the most popular GUIs are Unity, Gnome and KDE.[19] Each of these GUIs has its own desktop environment. However, because they more or less follow the same general principles, if you can use one, it's not hard to learn how to use another.

When you use a CLI or a TUI, you control a program by typing at it. With a GUI, you do use your keyboard, but mostly to enter data. Generally, you control your programs by using a pointing device, such as a mouse, trackball, touchpad, or pointing stick. For example, you would use a pointing device to pull down menus and make a selection; to manipulate objects by dragging and dropping; and so on.

Now back to Emacs. Every time Emacs starts it checks what type of interface you are using and acts appropriately. If you start Emacs while you are working within a GUI, Emacs runs as a graphics-based program inside a window. If you start Emacs from a text-based terminal, Emacs takes over the terminal's screen and runs as a TUI program. Nevertheless, no matter what system you are using (Unix, Windows, or Macintosh) and regardless of how you run the program, Emacs always works the same. Here is why.

Like so many other enduring programs, Emacs is a product of the times during which it was created. To be sure, over the decades, Emacs has been rewritten more than once and enhanced many, many times. Still, the program remains surprisingly close to its roots, and if you really want to understand it you need to have a sense of the historical conditions under which it arose.

Emacs was designed to work well with the hardware and software limitations of its time and, in the environment in which Emacs was developed, people interacted with programs by typing one command after another.

Realize then, that when you use Emacs today you are, literally, copying the keystrokes of programmers who, decades ago, decided which key combinations worked best for what they wanted to do. This is important to remember so when you set out to learn how to use Emacs, you don't think that what you are learning is unnecessarily complicated and arbitrary. Emacs is well-designed and, I promise you, you will see that, but it will take a while. Along the way, if you ever get discouraged

[19] See Personal Note #20, "KDE and Gnome" (Appendix A).
[20] Compared to your phone or tablet, which is easy to learn but hard to use.

or confused remember what I told you in Section 1.1: Emacs is difficult to learn, but easy to use.[20]

For this reason, no matter where and how you run Emacs, you will always control the program by typing at it — and the key combinations will always be the same. This means that the Emacs skills you learn from this book will last you forever.[21] That's the way it was designed back in the late 1970s (see Section 1.4), and the design still works well. Emacs is, arguably, the best TUI-based program ever written, and there is no need to change it. Indeed, changing Emacs from a text-based, command-driven program that requires you to learn a huge number of key combinations into an "easy-to-use", GUI-based program with icons and menus would be like sending Donald Duck to charm school.[22]

Section 2.6: Using a Unix Terminal

So far, we have discussed why the terminal/host system is a basic part of Unix (Section 2.3) and why you need a terminal to work with text-based programs (Section 2.5). The most important such program is the shell, the command processor that reads and interprets your commands (Section 2.2). Because the shell is a text-based program, you need to have access to a terminal in order to use Unix commands. You also need access to a terminal to use Emacs.

The specific details I am about to show you below are based on Ubuntu Linux, a very popular Linux distribution. However, most of what you read here will work with all Linux systems, and the general ideas apply to any type of Unix.

As we have discussed, no one uses real terminals anymore. Instead, we use terminal emulators. So whenever you hear people talk about a "terminal", you can assume that they are referring to a terminal emulator, not a real terminal. For example, if I say, "I am going to show you how to use a Linux terminal", what I mean is, I am going to show you how to use a Linux terminal emulator.

There are two different types of terminal emulators: virtual terminals and terminal windows. When you use a VIRTUAL TERMINAL, the terminal emulator uses your entire screen to provide you with a totally text-based experience. When you use a TERMINAL WINDOW, the terminal emulator runs in a window within a GUI (graphical user interface).[23]

Note For historical reasons, you will sometimes see a virtual terminal referred to as a VIRTUAL CONSOLE.

---

[21] Or until you die, whichever comes first.
[22] It's not going to work, and all you are going to do is annoy the duck.
[23] See Personal Note #21, "Aren't All Terminals Virtual?" (Appendix A).

Linux systems come with six built-in virtual terminals (and you can create more if you need them). To access the six built-in terminals, you use the key combinations shown in Figures 2-4 and 2-5.

Terminal	Key Combination
1	<Ctrl-Alt-F1>
2	<Ctrl-Alt-F2>
3	<Ctrl-Alt-F3>
4	<Ctrl-Alt-F4>
5	<Ctrl-Alt-F5>
6	<Ctrl-Alt-F6>

FIGURE 2-4. Accessing a Virtual Terminal From the GUI.

To switch from the GUI to a specific virtual terminal, use <Ctrl-Alt-F1> through <Ctrl-Alt-F6>. To return to the GUI, use <Alt-F7> or <Ctrl-Alt-F7>.

Terminal	Key Combination
Next	<Alt-Right>
Previous	<Alt-Left>
1	<Alt-F1> or <Ctrl-Alt-F1>
2	<Alt-F2> or <Ctrl-Alt-F2>
3	<Alt-F3> or <Ctrl-Alt-F3>
4	<Alt-F4> or <Ctrl-Alt-F4>
5	<Alt-F5> or <Ctrl-Alt-F5>
6	<Alt-F6> or <Ctrl-Alt-F6>
Return to GUI	<Alt-F7> or <Ctrl-Alt-F7>

FIGURE 2-5. Changing From One Virtual Terminal to Another.

To switch from one virtual terminal to another, you have several choices. <Alt-Right> changes to the terminal with the next highest number. <Alt-Left> changes to the terminal with the previous number. <Alt-F1> through <Alt-F6> changes to a specific terminal. Finally, <Alt-F7> will return you to the GUI (the Desktop Environment). Please note that, for convenience, the function key combinations can use either <Alt> or <Ctrl-Alt>.

To use a virtual terminal, you must log in by typing your userid and your password. To log out, you can either use the **`logout`** command, or press <Ctrl-D> to send an **`eof`** signal. (We will discuss signals and **`eof`** in Section 2.11.)

The advantage of using a virtual terminal is that it looks and works pretty much the same as if you were using a real terminal. Specifically, the display is all text and

uses the entire screen, which makes the characters easy to read. Moreover, there are no windows or icons or other elements to distract you, so it is easy to focus on one thing at a time.

You can switch from one virtual terminal to another by pressing the appropriate key combination.[24] For example, let's say you are working with the GUI and you want to use virtual terminal #1. You would use:

<Ctrl-Alt-F1>

After a while, you want to pause what you are doing to use a different virtual terminal. To switch to virtual terminal #2:

<Alt-F2>

Later, you decide to switch back to virtual terminal #1. Either of the following will work:

<Alt-Left>
<Alt-F1>

Finally, you decide to switch back to the GUI (which will be exactly how you left it):

<Alt-F7>

The second way to access a terminal is by using a terminal window. Unlike virtual terminals, terminal windows run within the GUI in their own window, like regular programs. The advantage of using a terminal window is that, as you work, you are within the GUI, so you can use all the other GUI-based tools that are part of your desktop environment. You can also have multiple windows open, each with its own program. If you want to work with more than one terminal at a time you can open more than one terminal window and, because everything is running within the GUI, you can copy and paste between a terminal window and any other window. (This is not the case with virtual terminals. Because each virtual terminal runs in its own environment, you can't copy and paste from one virtual terminal to another; and you can't copy and paste from a virtual terminal to the GUI, or from the GUI to a virtual terminal.)

Starting a terminal window is easy. However, before I give you the instructions, I want to take a moment to give you a brief introduction to the Ubuntu GUI and the default desktop environment, named Unity (see Section 2.5).

The Unity desktop environment (see Figure 2-6) is particularly easy to use. The column of icons on the far left (under the words "Ubuntu Desktop") is called the LAUNCHER. You use the Launcher to start programs. The top icon in the Launcher is the Ubuntu logo. You click it to open the DASH, a tool that lets you search for anything you want.

[24] If your function keys don't respond properly, check your keyboard. You may have to hold down a special key (such as <Fn>) to make your function keys work.

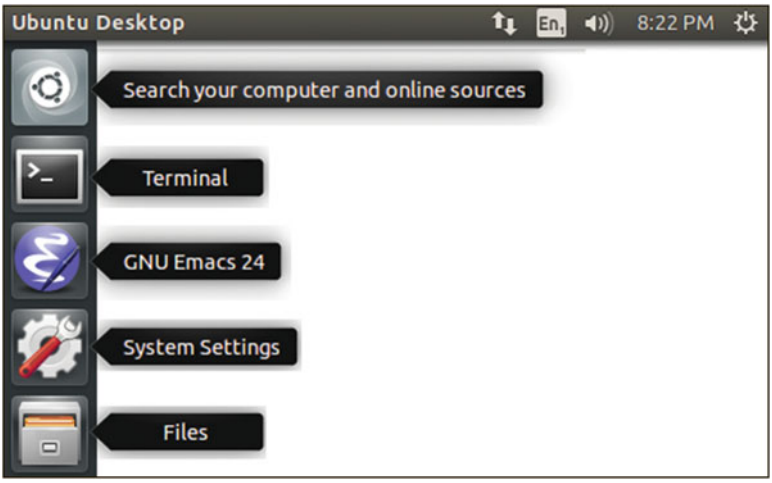

FIGURE 2-6. The Ubuntu Desktop.

The default Ubuntu desktop environment, Unity, is particularly easy to use. The column of icons on the left, under the words "Ubuntu Desktop", is called the Launcher. It is used to start programs. For example, to start a terminal window, just click on the Terminal icon. To start Emacs (once it is installed), click on the Emacs icon. The top icon in the Launcher is the Ubuntu logo. You can click it to open the Dash, which enables you to search for anything you want.

The easiest way to start a terminal window is to click on the Terminal icon in the Launcher (see Figure 2-6). Clicking on this icon will open a program (a terminal emulator) called TERMINAL inside a window of its own. If you want to start another, separate terminal window, just right-click on Terminal Icon and select "New Terminal". In this way, you can start as many terminal windows as you want.

If for some reason you don't see the Terminal icon in the Launcher, it is easy to find. Click the top icon on the Launcher to open the Dash and then type "terminal". You will see the Terminal icon, which you can click to start the program.[25]

■ **Note**

If the Launcher doesn't already have a Terminal icon, it is easy to put one there. Use the Dash to find and start the Terminal program. Once it starts running, its icon will appear in the Launcher. However, it will disappear when the program ends.

To make an icon stay in the Launcher permanently, right-click on the icon and select "Lock to Launcher".

To delete an icon from the Launcher, right-click on the icon and select "Unlock from Launcher".

Once an icon is in the Launcher, you may want to change its position. To do so, use your mouse and — in one smooth motion — drag the icon away from the Launcher, then up or down to the place you want to move it, and then back into the Launcher. (After you do it once, it's easy.)

---

[25] See Personal Note #22, "Ubuntu Terminal Emulators" (Appendix A).

Section 2.7: The Unix Command Line

The COMMAND LINE is a facility provided by an operating system that enables you to enter commands, one at a time. These commands are then processed by the shell (see Section 2.2). The command line is important because it enables you to use many of the tools that come with your operating system. (These tools are the "utilities" we talked about in Section 2.2.) When you use Unix, these tools are referred to as UNIX COMMANDS. With Windows, they are called WINDOWS COMMANDS.

As a smart person who wants to use his or her computer well, it behooves you to master the intricacies of the command line. In addition, it will help you to understand how to use the more important Unix commands (tools). As an Emacs user, it is especially important, because you can access the command line from within the Emacs environment, which makes it possible to use any of the commands you want — and there are hundreds of them — without having to leave Emacs. Before that can happen, however, you need to understand how to use the command-line on its own.

To use the command line well requires a lot of knowledge and takes a lot of experience, so I can't teach you everything right now. What I can do is make sure you know the basics, at least enough to get you started. If you are an experienced programmer, you may already know what I am about to explain. If so, it's probably better to not skip this part of the book. (Just read it quickly. Who knows, you may find something that is new to you.)

In the next few sections, I am going to show you how to use the Unix command line. What you are about to read will work for any type of Unix, including Linux and Mac OS X. If you use Windows, the basic ideas will be the same, but the details and the commands will be different.[26]

To access the command line, you must be using a terminal. If you are using Linux, just follow the directions in Section 2.6 and open either a terminal window or a virtual terminal. With a terminal window, you will be logged in automatically. With a virtual terminal, you will have to enter your userid and password to log in manually.

What's in a Name?

Command Line

In Unix, the term "command line" is used in two different, but related, ways.

First, "command line" refers to the facility that lets you enter Unix commands to be processed by the shell. Example: "To learn how to use Unix well, you need to master the command line."

Second, when you use a terminal, the bottom line of the display is called the "command line". This is because, as you type a command, this is where you see the characters you are typing.

---

[26] See Personal Note #23, "How to Access the Command Line with Mac OS X and Windows" (Appendix A).

Section 2.8: The Shell Prompt

As you use a terminal, the shell will indicate when it is ready for you to enter a command. It does this by displaying a special series of characters called the SHELL PROMPT. Every shell has a default prompt, which you can change if you want. Here is an example using Bash on one of my Linux systems:

```
harley@kajsa:~$
```

Although you can't see it, there is a space after the **$** (dollar sign) character. The space is there to create a bit of room between the prompt and the command you will be typing. For example, if you were to type the **date** command (to display the time and date), it would look like this:

```
harley@kajsa:~$ date
```

By convention, the second-last letter of the prompt (before the space) identifies the type of shell. Broadly speaking, there are two main families of shells in common use: the Bourne Shell family and the C-Shell family. If your shell is a member of the Bourne Shell family, you will see a **$** character, as in the examples above. If your shell is a member of the C-Shell family, you will see a **%** (percent-sign) character:

```
harley@kajsa:~%
```

If you are logged in as superuser (see Section 2.3), no matter what shell you are using you will see a **#** (number sign) character:

```
harley@kajsa:~#
```

A prompt that ends with # is telling you to be very careful: you are superuser and, if you make a mistake, you could cause a lot of damage.

■ **Note**

The most popular shells in the Bourne Shell family are Bash (**bash**) and the Korn Shell (**ksh**) in that order. The most popular shell in the C-Shell family is **tcsh** (pronounced "tee-see-shell").

The default shell for Linux systems and for Mac OS X is Bash. The default shell for BSD-based system is either **tcsh** or the Korn Shell. If you don't like your default shell, you can change it by using the **chsh** (change shell) command.

The first shell prompt I showed you is the default Ubuntu Linux prompt. Let's take a moment to dissect it:

```
harley@kajsa:~$
```

First, you see the name of my userid (**harley**). Then you see the **@** (at sign) character, followed by the name of my computer (**kajsa**), and a : (colon). Following the colon is a **~** (tilde) character. (It has a special meaning that I will

get to in a moment.) Next, you see a **$** (dollar sign) character. This indicates that your shell is a member of the Bourne Shell family (in this case, Bash). Finally, at the very end of the prompt is a space, which you can't see.

Note In the default Bash shell prompt, the **@** (at sign) and **:** (colon) characters are delimiters. A DELIMITER is an otherwise meaningless character that divides a string of characters into sections. For example, in the phone number 202-456-1414, the hyphens are delimiters.

So what is the meaning of the **~** (tilde) character? It is actually the pathname of your "current directory". (We will discuss this in Section 2.17.)

At any time, you can think of yourself as working within a directory. For now, I'll just tell you that the tilde stands for your personal "home directory". As you move from one directory to another, this part of the prompt will change to show where you are. For example, let's say that I changed from my home directory to the **Documents** subdirectory. The shell prompt would become:

```
harley@kajsa:~/Documents$
```

And let's say that this directory had a subdirectory of its own called **archive**. If I change to that subdirectory, the shell prompt would become:

```
harley@kajsa:~/Documents/archive$
```

Finally, if I changed to the **/usr/include/protocols** directory, the prompt would change to:

```
harley@kajsa:/usr/include/protocols$
```

To review, the general format of the default Bash shell prompt with Ubuntu Linux is:

*userid***@***computer* **:** *current-directory***$**

Section 2.9: What Unix Commands Look Like

Whenever you see the shell prompt, you can enter commands one at a time. When you are finished typing a command, you tell the shell to process it by pressing the Enter key (or the Return key on a Macintosh keyboard). As an example, here's the command that you would use to start Emacs (once it is installed) to edit a file named **important**:

```
emacs important
```

As a general rule, most Unix commands have the following format:

command-name short-options long-options parameters

In the example above, the command name is **emacs**; there are no short options or long options, and there is one parameter, **important**.

Short options start with a single **-** (hyphen) character. Long options start with two hyphen characters in a row. Let us analyze the following example using the **ls** command lists (used to display information about files).

```
ls -l --classify Documents Music
```

The command name is **ls**. There is one short option (**-l),** one long option (**--classify**), and two parameters (**Documents** and **Music**).

If a command has more than one short option, you can put them together with a single hyphen. For example, let's say you want to use the **ls** command with three short options (**-a -l -d**). The following commands are equivalent:

```
ls -ald
ls -a -l -d
```

When we talk about such commands out loud, there are two conventions we follow. First, for short, two-letter commands, we pronounce the separate letters. For example, when we talk about the **ls** command, we call it the "L S" ("ell-ess") command; and the name of the **vi** text editor is pronounced "V I" ("vee-eye").

Second, when we talk about options, we pronounce the **-** (hyphen) character as either "minus" or "dash". For example, let's say you wanted to tell someone to use the following command:

```
ls -l
```

You would say "Use the L S minus L command" or "Use the L S dash L command".

■ **Note** Unix considers upper- and lowercase letters to be completely different characters. If you are used to Windows or Mac OS X, remember this when you type commands. For example, the **-a** option is completely different from the **-A** option (see Section 2.18).

Section 2.10: Making Corrections as You Type Commands

As you type a Unix command, there are keys you can use to modify the command or to make corrections. The most important is the Backspace key. As you type, whenever you make a mistake, just press <Backspace> as many times as necessary and fix the mistake.

You can also make changes by using the two arrow keys, <Left> and <Right>, to move the cursor within the command. You can then use <Backspace> to delete the character to the left of the cursor, or <Delete> to delete the character at the cursor.

If you are used to a PC keyboard you probably already know how to use these keys. However, Unix also uses two other keys you may not know about. To delete the previous word, press <Ctrl-W>. To delete the entire line, press <Ctrl-U> (on some systems, <Ctrl-X>). For a summary of these keys, see Figure 2-7.

Key	Function
<Left>	Move cursor left one position
<Right>	Move cursor right one position
<Backspace>	Delete character to the left of the cursor
<Delete>	Delete character at the cursor
<Ctrl-W>	Delete the previous word
<Ctrl-U>	Delete the entire line (on some systems <Ctrl-X>)

FIGURE 2-7. Keys to Make Corrections When Typing a Command.

As you enter a command, here are the most important keys you can use to make corrections. These keys work with all Unix systems, including Mac OS X. The first four keys also work when you type commands with Microsoft Windows. However, <Ctrl-W> and <Ctrl-U> do not work with Windows, as they are Unix keys.

Note

On a Macintosh keyboard, there is no Backspace key. To delete to the left, you use the primary Delete key at the top-right corner of the main part of the keyboard. To delete at the cursor, you use the secondary Delete key, the one to the left of the End key.

The secondary Delete key is available only with a full-sized keyboard. If you have a compact keyboard, you won't have this key. Instead, use <fn-Delete> or <Control-D>.

Section 2.11: Two Important Keys: <Ctrl-C> and <Ctrl-D>

To enter a command, you type it and then press the Enter key. When you press <Enter>, you are telling the shell to process the command.

A small number of commands are built in to the shell. When you enter such a command, the shell handles it on its own. Most commands, however, require the shell to run another program on your behalf. For example, if you enter the **`date`** command (to display the time and date), the shell will start a program named **`date`**. The shell then places itself on hold, waiting for the program to finish. The moment the program finishes, the shell regains control. It then displays the shell prompt and waits for you to enter another command.

The same thing happens when you enter the **`emacs`** command. The shell starts Emacs and then puts itself on hold. When you quit Emacs, the shell regains control and displays the shell prompt.

From time to time you may want to quit (stop) a program that is running. How you do so depends on which user interface the program uses.

Programs that use a TUI (text-based user interface) take over the screen and create their own environment. Such programs will have a quit command that you use to end the program. For example, when you run Emacs, it will keep control until you stop it deliberately. (The most common way to do this is by pressing <Ctrl-X> <Ctrl-C>.) Here is another example. In Section 2.11, I will show you how to access the Unix manual. If you are displaying information from the Unix manual and you want to stop, you use the **q** (quit) command.

Programs that use a CLI (command-line interface) are different. They don't take control of the screen. Instead, they write each line of output to the bottom of the screen, causing all the other lines to scroll up.

With many commands, the output is short and the command finishes quickly. This is the case, for example, when you enter the **date** command. Within an instant, the shell runs the date program, the time and date are displayed, the program ends, and you see the next shell prompt.

Some CLI programs, however, may not stop on their own, either by design or if they run into a problem. When this happens, you will have to interrupt the program to get it to stop. You do so by sending it a SIGNAL, a special notification that is sent to a program that is running. Specifically, to stop a program, you press <Ctrl-C> to send the program an **intr** signal (INTERRUPT SIGNAL). Most CLI programs will stop instantly when they detect an **intr** signal.

If you want to try this, enter the following command:

```
ping www.harley.com
```

The **ping** command sends a PING SIGNAL to another computer to see if it responds, and then displays technical information about the response (or lack of response). If you use **ping** without any options, it will keep sending the signal and displaying output indefinitely. To stop **ping**, simply send an **intr** signal by pressing <Ctrl-C>.

■ **Note**

When you are working with Emacs, <Ctrl-G> acts a lot like <Ctrl-C> does with CLI-based programs. Pressing <Ctrl-G> won't stop Emacs (because it is a TUI-based program), but it will cancel any action you may have started within Emacs.

We will discuss this later, so don't worry about it right now. Just file it away as a useful piece of information that will come in handy one day: <Ctrl-G> with Emacs is similar to <Ctrl-C> with the Unix command line.

Some CLI programs invite you to enter data for the program to process, and as long as you keep entering data the program will keep running. An interesting example is the **factor** program. Once you start **factor**, it waits for you to enter a whole number. It then breaks the number down into prime factors, which it displays.

For example, if you enter the number 68, **factor** will calculate that 68 = 2 x 2 x 17, and respond with:

```
68: 2 2 17
```

At this point, the program waits for you to enter another number. On its own, **factor** will never stop. To stop such programs, you must indicate that there is no more data. To do so, you press <Ctrl-D> to send an **eof** signal (END OF FILE SIGNAL).

If you want to try this, enter the command:

```
factor
```

Then enter any whole number you want followed by <Enter>, over and over, as long as you want. When you want to stop, press <Ctrl-D>.

As we have discussed, when you use the shell, you enter one command after another. The shell processes each command appropriately and then waits for you to enter another one. Because the shell is a CLI program, you can press <Ctrl D> to indicate that there is no more data. When you do this, the shell, like all CLI programs, will stop. The moment the shell stops, your userid is logged out automatically.

Thus, when you are using a terminal, whenever you see a shell prompt, you can log out by pressing <Ctrl-D>. (You can also enter the **logout** command.)

Note If a CLI program is waiting for data and pressing <Ctrl-D> doesn't stop the program, you can press <Ctrl-C> and send it an **intr** (interrupt) signal. Most well-behaved programs will then quit. (Of course, programs are like people. Not all of them are well behaved.)

Section 2.12: The History List; Command Line Editing

As you use the command line, the shell maintains a list of the commands you enter. This list is called the HISTORY LIST. At any time, you can recall commands from the history list, which you can then modify and reuse.

For example, let's say you want to use the **ls -l** command to display information about the files in the directory named **/usr/include/protocols**. The command you want is:

```
ls -l /usr/include/protocols
```

However, by accident, you enter:

```
ls -l /usr/include/protcols
```

This won't work because you forgot to type the second "o" in **protocols**.

Instead of typing the entire command again (and probably making another mistake), you can recall what you just typed from the history list. It will then be the work of a moment to fix the mistake and re-enter the corrected command. (I'll show you how in a moment.)

You can do a lot with the history list and the details are complex. However, because it is worth learning, I will describe a few of the basic techniques. Afterwards, I will explain how this is all connected with Emacs.

Think of the history list as an invisible file containing your most recent Unix commands. Each time you enter a new command, it is inserted into the bottom of the history list. So the last line of the invisible file is always the last command you entered.

The simplest way to move up and down through the invisible file is to use the <Up> and <Down> (arrow) keys. As you do, the shell displays the command to which you are pointing within the invisible file. When you press <Up>, the current command vanishes and is replaced by the previous command, as if you were going back in time. If you go too far, you can press <Down> to move back down the list.

Once you display a command from the history list, you can modify it however you want. You then press <Enter> to tell the shell to process the newly modified command. Let's consider our example.

To start, let's say you enter the following, incorrect **`ls`** command:

```
ls -l /usr/include/protcols
```

Because you made a spelling mistake, you see the following error message:

```
ls: cannot access /usr/include/protcols: No such file or
directory
```

To recall the incorrect command from the history list, press the <Up> arrow once. You now see, once again, the incorrect command:

```
ls -l /usr/include/protcols
```

To fix it, press <Left> four times to move the cursor between the **`t`** and the **`c`** in **`protcols`**. Then press **`o`** to insert the letter "o". You now see:

```
ls -l /usr/include/protocols
```

To enter the corrected command, press <Enter>.

As I said, there is a lot you can do with the history list and it is all worth learning. However, it will take you some time to learn. At the very least, however, here are the basic ideas I want you to remember:

1. The history list is an invisible file containing your recently entered Unix commands.
2. To move up and down through the list, use the Up key and the Down key. As you move through the history list, the shell will show you the appropriate command.
3. To move the cursor within a command, use <Left> and <Right>.
4. As you make corrections and changes, you delete characters by using <Backspace>, <Delete>, <Ctrl-W>, and <Ctrl-U> (or <Ctrl-X>). For a summary of these keys, see Figure 2-8.

Unix Keys	Function
<Up>	Display previous line in history list
<Down>	Display next line in history list
<Left>	Move cursor left one position
<Right>	Move cursor right one position
<Backspace>	Delete character to the left of the cursor
<Delete>	Delete character at the cursor
<Ctrl-W>	Delete the previous word
<Ctrl-U>	Delete the entire line (on some systems <Ctrl-X>)

Emacs Keys	Function
<Ctrl-P>	Display previous line in history list
<Ctrl-N>	Display next line in history list
<Ctrl-A>	Move cursor to beginning of the line
<Ctrl-E>	Move cursor to end of the line
<Ctrl-B>	Move cursor left (backward) one position
<Ctrl-F>	Move cursor right (forward) one position
<Alt-F>	Move cursor right (forward) one word
<Alt-B>	Move cursor left (backward) one word
<Ctrl-K>	Delete from cursor to end of the line
<Ctrl-A> <Ctrl-K>	Delete the entire line

FIGURE 2-8. Key Combinations to Use When Typing a Command.

The top list shows the keys we discussed in Section 2.10 that you can use to make corrections when you are typing a command. The bottom list shows some of the key combinations copied from Emacs as part of the GNU Readline facility. You can use the keys in both these lists to move within the current line and the history list, and to help you make changes.

The shell has a powerful, built-in facility called COMMAND LINE EDITING. In fact, it is command line editing that enables you to modify commands as you type, to access the history list, and to use autocompletion (which we won't go into here).

You will recall that, in the last few sections, I showed you how to enter commands and make corrections as you were typing. I then explained how to move within the history list, recall a previous command, modify it, and enter it again. What I was showing you was actually basic command line editing. What you didn't realize is that I was also showing you basic Emacs editing commands. (So if you were following along on your own computer, you were actually using a few Emacs keys.)

This is not as strange as you might think. The Emacs editing commands are so powerful and (once you master them) so easy to use, that many other programs also use them. In particular, the most popular shells (Bash, **`tcsh`**, Korn Shell) all recognize Emacs key combinations that you can use for command line editing, that is, for moving through the history list and modifying commands, and for autocompletion. These capabilities are provided by a software library called GNU Readline. For reference, some of these Emacs key combinations are shown in Figure 2-8.

This means that, as you learn the Emacs key combinations, you can immediately use them with the shell for command line editing: just another reason why, the more you learn Emacs, the closer you are to becoming a totally fulfilled and actualized human being.

However, none of this is a substitute for the real thing, so let's move on. We still have more to cover before we can actually start using Emacs.

Section 2.13: The Unix Manual

All Unix commands and programs are documented in the UNIX MANUAL, a part of every Unix system. When a Unix person talks about THE MANUAL, they always mean the Unix manual. For example, if you ask your grandmother how to use the **`ls`** command, she might tell you, "Look it up in the manual, and let me know if you have any questions."

The Unix manual is considered to be authoritative, the definitive reference when it comes to basic information about a particular command or program. As such, when a programmer changes a program — say, to add a new option — he or she is expected to update the entry for that program in the manual.

Each entry in the Unix manual is called a MAN PAGE. Don't take this term literally, though: most man pages are much longer than the size of a printed page. (Here is an extreme example: if you printed the Bash man page, a particularly long one, it would take 155 printed pages.)

To display a man page, you use the **`man`** command. Type **`man`** followed by the name of the command or program you want to learn about. For example, to read the man page for the **`ls`** command, enter:

```
man ls
```

To learn about the **`man`** command itself, use:

```
man man
```

To look up something on the Bash shell man page, use:

```
man bash
```

■ **Note:**

The Unix manual is a *reference* manual, not a teaching guide. At a minimum, each page shows you:

- Name of the command
- Description (usually only a few words)
- Syntax summary
- Options (description of all the options)

The SYNTAX of a command is the formal description of all the different ways you can enter the command and its options.

Section 2.14: Using the less Pager Program

When you use a terminal to display man pages, you are working in a text-based environment using a CLI (command line interface). As such, the **man** command writes its output to the bottom line of the terminal screen, causing previous output to scroll up. This works fine if the man page is short enough to fit on your screen. However, most man pages are much too long. Without a tool to display the output in a controlled fashion, most of the text would scroll off the top of the screen so quickly, you wouldn't be able to read it.

To make the output readable, the **man** command does not write it directly to the screen. Instead, it sends it to a special tool called a PAGER PROGRAM or, more simply, a PAGER. The job of a pager is to take all the output and display it one screenful at a time. This lets you read the output in a controlled fashion. In fact, you can use the pager to move backwards and forwards, to search for a particular string of characters, to jump to the beginning or the end of the page, and much more.

The default Unix pager is called **less**.

What's in a Name?

less

The original Unix pager was a simple program named **more**, written in 1978 by Dan Halbert, then a grad student at U.C. Berkeley. Halbert wrote **more** to display text one screenful at a time. If there was more to come, it would display a message at the bottom of the screen:

--More--

Using **more** was easy. After you read what was on the screen, you would press <Space> to display more. Thus, you could read an entire file, one screenful at a time, simply by pressing the <Space> bar.

The biggest problem with **more** is that it could only move forward. In 1983, a programmer named Mark Nudelman was working with large log files, and he needed a pager that could move forward and backward. He and his colleagues used to joke that what they needed was a "backwards **more**", so when he wrote a replacement for **more** that could move backwards, he called it **less**.

When **less** is running, it controls the entire screen. It starts by displaying the first screenful of output, with an informative message on the bottom line of the screen (the command line). It then waits for you to enter a command, of which there are many.

Note

It is interesting to note that, the moment **less** takes control of the screen, it changes your experience from using a command-line (CLI) interface to using a text-based interface (TUI). That is what enables it to use the entire screen in order to control the flow of the output. Emacs does exactly the same thing, as do many other programs.

For a discussion of the three basic user interfaces (CLI, TUI, GUI), see Section 2.5.

As I said, **less** has many commands that you can use while you are reading text. If you want to see them all, look at the **less** man page:

```
man less
```

(As you do, you will be using **less** to read about **less**.)

For reference, Figure 2-9 shows the basic **less** commands. The three most important ones are:

- Press the <Space> bar to display the next screenful of text
- Press the **q** key to quit the program
- Press the **h** key to display help information

Notice that you can navigate by using either single-letter commands, or by using the special keys <Home>, <End>, <PageDown>, and <PageUp>.

When you have a moment, enter the **man** command above and spend some time practicing with **less.** This is a program that you need to learn how to use, because it is the key that unlocks the Unix manual.

Letters	Function
h	Display help information
q	Quit the program
g	Go to first line of text
G	Go to last line of text
<Space>	Move forward (down) one screenful
b	Move backward (up) one screenful
/*pattern*	Search forward for specified pattern
?*pattern*	Search backward for specified pattern
n	Repeat search in the same direction (next)
N	Repeat search in the opposite direction
!*command*	Run the specified shell command

Special Keys	Function
<Home>	Go to first line of text
<End>	Go to last line of text
<PageDown>	Move forward (down) one screenful
<PageUp>	Move backward (up) one screenful

FIGURE 2-9. Commands to use with less.

*The default Unix pager program is named **less**. The purpose of **less** is to display text, one screenful at a time. While you are reading, there are many commands you can use. Here are the most important ones.*

Section 2.15: The Three Types of Unix Files

As we discussed in Section 1.2, the primary use of Emacs is to create and modify text files: programs and scripts, data files, HMTL files, LaTeX documents, logs, and so on. For this reason, it is important to understand how files are organized. Specifically, I want to make sure you are familiar with the idea of a hierarchical filesystem, as well as the basic concepts that support it: files, directories, and subdirectories. I also want you to know the names of the Unix commands we use to work with files and directories.

Being able to manage your files is a basic skill you need if you are to use Emacs well. To do so, there are two sets of tools, and I want you to be skillful with both of them. First, you can use the standard Unix file and directory commands that everyone learns. Second, you can use a TUI-based file management program called Dired (Directory Editor) that is part of Emacs. (We will talk about Dired in Section 12.6.) In either case, you will need to know the basic concepts, so let's start with files.

There are three types of Unix files: ordinary files, directories, and pseudo files. An ORDINARY FILE is what most people think of when they use the word "file": a collection of data that has a name and is stored on a disk or other storage device. With Emacs, we generally work with text files, ordinary files that contain the characters you can type on a keyboard: letters, numbers, punctuation, spaces and tabs.

A DIRECTORY is used to organize groups of files. For instance, you might create a directory named **`circus`** to hold a collection of photos from a trip to Centerboro, New York, where you saw Boomschmidt's Stupendous and Unexcelled Circus.[27] When we talk about a directory, we say that it CONTAINS or HOLDS other files. Literally, this is not the case. Directories contain information about files, not the actual files themselves.

The reason directories are so important is that they can contain other directories, as well as ordinary files. This allows you to create a hierarchy of directories to organize your data. When a directory contains another directory, with respect to one another we call the first one the PARENT DIRECTORY and the second one a SUBDIRECTORY.

For example, if you have a lot of photos in your **`circus`** directory, you might organize them by using three subdirectories: **`animals`**, **`friends`** and **`martians`**. In this case, **`circus`** is the parent directory of **`animals`**, and **`animals`** is a subdirectory of **`circus`**.

Traditionally, the term "directory" is used by people who are comfortable working with a command-line interface. People who use a GUI (graphical user interface) tend to refer to directories as FOLDERS and subdirectories as SUBFOLDERS.

[27] See Personal Note #24, "Freddy and the Men From Mars" (Appendix A).

■ **Note**

If you ever decide to use an online dating service, I suggest you specify, "I prefer someone who organizes their directories well."

By using the word *directories* instead of *folders*, you will eliminate all the Windows and Mac users whose desktops are filled with icons.

A moment ago, I told you that when most people think about a file, they consider it to be "a collection of data that has a name and is stored on a disk or other storage device". This description is certainly accurate for both ordinary files and directories. As a definition, it is intuitive and for some systems, such as Microsoft Windows, it works just fine. With Unix, however, the concept of a file is much more generalized. Specifically, a Unix FILE is any source from which data can be read, or any target to which data can be written.

There are different types of Unix files, but they all fall into one of three categories: ordinary files, directories, or pseudo files. Unlike ordinary files and directories, PSEUDO FILES do not store data. Instead, they provide services to programs using the same methods that are normally used to read and write data from ordinary files. As such, pseudo files are part of the Unix filesystem and, like ordinary files, they are organized into directories. The three most important types of pseudo files are special files, named pipes, and proc files.

SPECIAL FILES (also called DEVICE FILES) represent physical or emulated devices. NAMED PIPES connect the output of one program to the input of another. PROC FILES provide technical information about the system itself.

Although we don't need to get into a lot of details, it is important that you understand the general concept: pseudo files do not store data in the regular manner, but they can be used for input and output in the same way as you would use an ordinary file.[28]

■ **What's in a Name?**

File

Strictly speaking, a Unix file is any input source or any output target. As we have discussed, there are three types of files: ordinary files, directories, and pseudo files (which include device files). However, when people talk about ordinary files, they rarely bother to use the word "ordinary". However, the meaning should be clear from the context.

For example, if I were to tell you, "Files and directories require storage space, I am referring to ordinary files. However, if I say, "You can use the **`ls`** command to display a list of the files in a directory," I am referring to any type of file, including ordinary files, subdirectories, and pseudo files.

One last example. The title of Section 2.18 is "File and Directory Names". I hope it is clear from the context that, when I wrote the title, I was referring to ordinary files and directories.

If this idea is new to you, I suggest that whenever you see the word "file" used, you take a moment to ask yourself: In this context, does the word "file" refer to any type of file, or just to ordinary files? After a while, it will become second nature, and you will just know.

[28] See Personal Note #25, "Special Files and Proc Files" (Appendix A).

Section 2.16: The Tree-Structured Filesystem

Most people run Emacs on some type of Unix system, such as Linux, FreeBSD, NetBSD, or OpenBSD. However, you can also use Emacs with Mac OS X (which is actually Unix) and Microsoft Windows (which is most definitely *not* Unix). There are some differences between how filesystems work with Linux and BSD compared to OS X and Windows, and we will talk about them in Section 2.17. What's most important, however, is that all of these operating systems organize files into a hierarchy, a basic paradigm I want to make sure you understand.

A typical Unix system contains well over 300,000 files. (I am not exaggerating.[29]) These files are organized into a TREE-STRUCTURED FILESYSTEM in which one main directory, called the ROOT DIRECTORY, serves as the parent directory for the entire filesystem. The root directory is — directly or indirectly — the parent of all the other directories in the system. (Think of a tree where the trunk is the root directory, and all the branches are subdirectories.)

When we write the name of the root directory, we don't use the name "root". Instead, we simply write a single **/** (slash) character. Here is an example using the **ls** (list) command, which is used to display information about a directory. The following command displays the names of the files in the root directory:

```
ls /
```

The basic way to write the name of a directory is to simply trace the path from the root directory, through every subdirectory that leads to the target directory. Within the sequence of directories, we use **/** characters as delimiters, to separate one directory name from another. Here is an example.

Let's say the root directory contains a subdirectory named **home**, which contains a subdirectory named **harley**, which contains a subdirectory named **circus**, which contains a subdirectory named **animals**. We would write the full name of the **animals** directory as follows:

```
/home/harley/circus/animals
```

To display the contents of this directory, we can use the following **ls** command:

```
ls /home/harley/circus/animals
```

Note When you say directory names out loud, you pronounce the "slash". Thus, the directory name above is pronounced: "slash home slash harley slash circus slash animals".

If we want to reference the name of an ordinary file, we simply put another **/** at the end of the last directory and add the file name onto the end of the list.

---

[29] See Personal Note #26, "How Many Files Are on Your Unix System?" (Appendix A).

For example, let's say there is a file called **leo.jpg** in the **animals** directory. We can reference it as follows:

```
/home/harley/circus/animals/leo.jpg
```

When we use this pattern to specify a sequence of directories (possibly with a file at the end) separated by **/** characters, we call it a PATHNAME or, more simply, a PATH. If a pathname ends with a file, we call that part of the path a FILENAME. For instance, the example above is a pathname. The last part, **leo.jpg**, is a filename.

You will have noticed that our example used the directory **home**, which is a subdirectory of the root directory. Each time a userid is created, the system creates a HOME DIRECTORY for the person using that userid to store his or her personal files. Each userid's home directory is a subdirectory of **/home**, and its name is the same as the userid. For example, let's say that a system is used by four different people whose userids are **harley**, **lydia**, **jeff** and **dmitry**. Their home directories will be:

```
/home/harley
/home/lydia
/home/jeff
/home/dmitry
```

Each of these people can create their own subdirectories within their home directory. In this way, every Unix user is able to design a personal directory tree to organize his files in a way that make sense to him or her. As an example, let's say that, within the **/home/harley** directory, there is a **circus** subdirectory, and within that, three subdirectories named **animals, friends** and **martians**. We have the following personalized directory tree:

```
/home/harley
/home/harley/circus
/home/harley/circus/animals
/home/harley/circus/friends
/home/harley/circus/martians
```

The **/home** directory is only one of many built-in subdirectories of the root directory. There are also **/bin**, **/dev**, **/etc** and more. It is these "first-level" subdirectories that define the shape and function of the Unix filesystem. Many (but not all) of these subdirectories are the same from one Unix system to another. Once you get used to the basic plan, it will be generally the same no matter what system you use to run Emacs as long as it is Unix: including Linux, any BSD system, or Mac OS X. (Microsoft Windows, however, is different in that it uses a completely separate file system for each storage device.)

Note

If you want to see the directory organization used with your system, you look at the **hier** (hierarchy) man page:

man hier

Whenever you use a new system, take a few moments to look at the **hier** man page to see how the filesystem is organized.

Although there is no universal master plan that all Unix systems are required to follow, there is a widely recognized plan called the FILESYSTEM HIERARCHY STANDARD, or FHS, that is used with most Linux systems. (In fact, the FHS is maintained by the Linux Foundation.) For reference, Figure 2-10 shows the most important directories within the Filesystem Hierarchy Standard, Version 3.0.

Directory	Description
/	Root directory
/bin	Essential user commands (binaries)
/boot	Boot loader files
/dev	Device files (special files)
/etc	System configuration files
/home	User home directories
/lib	Essential shared libraries and kernel modules
/media	Mount point: removable media
/mnt	Mount point: temporarily mounted filesystems
/opt	Application programs ("optional" software)
/proc	Proc files
/root	Home directory for the root userid (superuser)
/run	Temporary data used by programs that are running
/sbin	System programs (binaries)
/srv	Data for system services
/tmp	Temporary files, not preserved between reboots
/usr	Sharable, read-only data
/var	Variable (changeable) system data

FIGURE 2-10. The Most Important Directories Within the Filesystem Hierarchy Standard.

All Unix systems organize files and directories into a tree-structured filesystem using a single root directory. The Filesystem Hierarchy Standard is the basic plan followed by Linux systems. To see the details for your particular system, look at the **hier** *man page.*

■ **What's in a Name?**

root, **`root`**

In Unix, the word "root" has two meanings. First, the root directory is the name of the top-level directory for the entire filesystem.

To help you with the metaphor, think of a Unix filesystem as a very, very large tree growing from a single root. The many levels of directories and subdirectories are the branches of the tree. The ordinary files are leaves, and the pseudo files are imaginary leaves. (If you want to see a visual representation of a Unix filesystem, look for a photo of a large *Tipuana tipu* tree.)

The second meaning: **`root`** is also the name of the userid for the superuser (see Section 2.3). The superuser userid was named after the most important directory in the filesystem.

Section 2.17: The Current Directory and Pathnames

At all times, one specific directory is designated as your CURRENT DIRECTORY, also called your WORKING DIRECTORY. Informally, this is the directory in which you are working at that moment. When you log in, your current directory is set to your home directory. For example, my userid is **`harley`**, so when I log in, my current directory is set to **`/home/harley`**.

In Section 2.16, we discussed how to specify pathnames. Starting from the root directory (**`/`**), you specify each subdirectory (or filename) using **`/`** characters as delimiters. For example:

```
/home/harley/circus/animals/leo.jpg
```

A pathname that begins with a **`/`** character starts from the root directory. This is called an ABSOLUTE PATHNAME. If the pathname doesn't begin with a **`/`**, the pathname starts from your current directory. This is called a RELATIVE PATHNAME. Here is an example.

Let's say that, at this moment, your current directory is **`/home/harley`**. The following two paths are equivalent:

```
/home/harley/circus/animals/leo.jpg
circus/animals/leo.jpg
```

Because the second pathname does not begin with a **`/`**, it starts from your current directory, which is **`/home/harley`**.

The concept of a current directory is important because it means that you rarely have to type long, absolute pathnames. Once you set your current directory to wherever you want to work, you can use shorter pathnames relative to that directory (as I did in the last example).

For convenience, there are three standard abbreviations you can use when you type pathnames. First, the tilde (**`~`**) character is a synonym for your home directory. For example, if your home directory is **`/home/harley`**, typing **`~`** at the beginning

of a pathname is the same as typing **/home/harley**. Thus, the following two **ls** commands would be equivalent. (The command **ls -l** displays information about a file.)

```
ls -l /home/harley/circus/animals/leo.jpg
ls -l ~/circus/animals/leo.jpg
```

The other two abbreviations are **.** (a single period) and **..** (two periods in a row), pronounced "dot " and "dot-dot" respectively. The **.** abbreviation stands for your current directory. For example, let's say that your current directory is:

```
/home/harley/circus/animals
```

The following two commands both refer to the file **leo.jpg** in this directory:

```
ls -l leo.jpg
ls -l ./leo.jpg
```

You might ask, why would you want to type a **.** character at the beginning of a pathname? Normally, you wouldn't because it's not necessary. However, there are times you do need to use it. For example, if you want to run a program from your current directory, and that directory is not in your search path, you must use **./** to specify the path explicitly. (If you don't understand what I just said, you can ignore it.)

The final abbreviation **..** stands for the parent directory. You use it when you want to go "up" one level within a pathname. For example, let's say that the **circus** directory has three subdirectories: **animals, friends** and **martians**. The **martian** directory contains a file named **chirp-squeak.jpg.** You want to display information about the file **chirp-squeak.jpg** in the **martians** directory, and your current directory is:

```
/home/harley/circus/animals
```

The following two commands are equivalent:

```
ls -l /home/harley/circus/martians/chirp-squeak.jpg
ls -l ../martians/chirp-squeak.jpg
```

The first command uses an absolute pathname that starts from the root directory. The second command uses a relative pathname that starts from the current directory (**animals**), goes up one level (to **circus**), and then goes down one level (into **martians**).

At this point, you may be wondering, at any moment, how do you know what directory is your current directory? There are two ways.

First, you can use the **pwd** command. The name stands for "print working directory":

```
pwd
```

Remember this command: if you ever get lost, **pwd** will tell you where you are.

■ **What's in a name?**

Print

As we discussed in Section 2.3, the oldest Unix terminals produced output by printing on a continuous roll of paper. Eventually, printing terminals were replaced by video terminals, and the idea of "printing data" changed into "displaying data". Nevertheless, it is still traditional among Unix people to use the term "print" to refer to displaying output.

For example, if you look at **pwd** man page, the description of this command is given as: "Print the full filename of the current working directory."

The second way to find out the name of your current directory is to look at your shell prompt. As we discussed in Section 2.8, it is likely that, by default, your shell prompt will display the name of your current directory. For example, the default Ubuntu Linux shell prompt is:

userid`@`*computer*`:current-directory`**`$`**

This means that, unless you have changed your shell prompt, it will always show you where you are in the directory tree.

Here are some examples using the **cd** command, which is used to change your current directory. (All you do is specify the directory to which you want to change.)

■ **Note** When you use the **cd** command without specifying a directory, it will, by default, change to your home directory. Thus, the following two commands are equivalent:

```
cd
cd ~
```

In the following examples, I will include the shell prompt so you can see how the current directory is changing. Before we start, I will tell you that my userid is **harley** and my computer is named **kajsa.**[30]

The examples start just after I log in, at which time my current directory is set to **/home/harley**. I then use **cd** to move from one directory to another. Take your time to read the examples slowly, and make sure you understand exactly what is happening. As you read, remember that **~** stands for the home directory, and **..** refers to the parent directory.

```
harley@kajsa:~$ cd circus
harley@kajsa:~/circus$ cd animals
harley@kajsa:~/circus/animals$ cd ../martians
harley@kajsa:~/circus/martians$ cd ~
harley@kajsa:~$ cd /usr/share/man/man1
harley@kajsa:/usr/share/man/man1$ cd ..
harley@kajsa:/usr/share/man$ cd ..
```

[30] After my dog Kajsa Anka, also known as Sadie.

```
harley@kajsa:/usr/share$ cd ..
harley@kajsa:/usr$ cd ..
harley@kajsa:/$ cd ..
harley@kajsa:/$ cd
harley@kajsa:~$ cd circus/martians
harley@kajsa:~/circus/martians$ cd ../..
harley@kajsa:~$
```

Notice that you can use more than one `..` in a row to go up more than one level at a time (second line from the bottom). Notice also that when you try to go up from the root directory nothing happens (fifth line from the bottom).

To finish this section, I have created two reference lists to show you the most important file and directory commands. Figure 2-11 shows the most important commands to use with ordinary files. Figure 2-12 shows the most important directory commands.

Command	Description
`cat`	Display a very short file
`cat`	Combine (catenate) multiple files
`chmod`	Modify (change) file permissions
`cmp`	Compare two files to see if they are the same
`cp`	Copy files
`du`	Display disk usage for files
`file`	Analyze file type
`find`	Search for files in directory tree, then process results
`head`	Display the beginning of a file
`less`	Display contents of file, one screenful at a time
`ls`	Display (list) information about files
`ls -l`	Display full information (long listing) about files
`mv`	Move files
`mv`	Rename files
`od`	Display contents of a binary file (octal dump)
`pwd`	Display name of current directory
`rm`	Delete (remove) files
`tail`	Display the end of a file
`touch`	When file does not exist: create brand new empty file
`touch`	When file exists: update access and modification times
`whereis`	Find files associated with a command

FIGURE 2-11. The most important file commands.

When you use Emacs there will be many times when you need to manipulate your files. Sometimes you can do it from within Emacs, but a lot of the time, it will be easier and faster to use the standard Unix file commands. These are the most important ones, and I recommend you learn how to use them all. (Notice that **`cat`**, **`mv`**, *and* **`touch`** *each perform two different functions.)*

Command	Description
cd	Change your current (working) directory
chmod	Modify (change) directory permissions
du	Display disk usage for directories
ls	Display (list) information about directories
ls -l	Display full information (long listing) about directories
mkdir	Create (make) a directory
mv	Move directories
mv	Rename directories
pwd	Display name of current directory
rmdir	Delete (remove) an empty directory
tree -d	Display a diagram of a directory tree

FIGURE 2-12. The most important directory commands.

Unix uses a very large, tree-structured file system. Within that file system, starting from your home directory, you can build a tree structure of your own in a way that suits you. This will become more and more important, as you develop a facility with Emacs. Here are the commands you can use to build, maintain, and use your own personal directory tree. (Notice that **mv** *performs two different functions.)*

Why do I list all these commands? Emacs is linked tightly to Unix and to the Unix command line, and understanding how to use files and directories is a basic skill you need to know. If you talk to anyone who is a skilled Emacs user, it is likely that he or she will know how to use all the commands I have listed in these two tables.

Section 2.18: File and Directory Names

To complete our discussion about Unix and its file system, I want to spend a few minutes talking about how to name files, specifically, ordinary files and directories. There are small differences with Mac OS X, and larger differences with Microsoft Windows, which we will talk about in Section 2.19.

Before we start, I want to take a moment to remind you that the topics we have covered in this chapter give you the basic knowledge about Unix you need to use Emacs. However, there is a lot more to learn about Unix, and it will take time and experience for you to do so. If you want a more comprehensive Unix reference, please get a copy of my book *Harley Hahn's Guide to Unix and Linux*.[31] This is a book you will want to keep as a permanent reference, so I recommend you look for an actual printed book, not an electronic copy.

Now, to continue. There are four important ideas you need to remember about naming files and directories.

[31] *Harley Hahn's Guide to Unix and Linux*, McGraw-Hill Higher Education, 2008. The ISBN is 0073133612.

1. LENGTH

File and directory names can be up to 255 characters long. However, unless you have a good reason to use a long name, stick with names that are short, so they will be easy to type.

2. CHARACTERS

When you name a file, choose from the following characters:

- Lowercase letters: **a b c**... **z**
- Uppercase letters: **A B C**... **Z**
- Numbers: **0 1 2 3 4 5 6 7 8 9**
- Hyphen: **-**
- Underscore: **_**
- Period: **.**

Unix is very flexible, and you can use almost any character you want in a file name, even spaces and tabs. However, don't do it. Your life will be a lot easier if you stick with the characters I listed above.

Note Remember, when you pronounce file names, the hyphen is called "minus" or "dash", and the period is called "dot".

If you end up with file names that have spaces, tabs, or most other special characters, you will have to put quotes around the names to make your commands work properly. For example, let's say you have created a directory named:

```
photos from the circus
```

Every time you use it in a command, you will need quotes around it:

```
ls -l 'photos from the circus'
```

If you don't use quotes, the shell will think you are referring to four separate files: **photos**, **from**, **the** and **circus**.

Note When you type a command and you want to indicate that certain characters are to be taken literally, you use either single quotes (like I did above) or double quotes ("like this"). For technical reasons, single quotes work better, so unless you are doing something that you know requires double quotes, get in the habit of using single quotes.

---

3. UPPER AND LOWER CASE

With respect to the letters of the alphabet, capital letters are called UPPERCASE, and small letters are called LOWERCASE. If a filesystem distinguishes between upper- and lowercase letters, we say that it is CASE SENSITIVE. If a filesystem does not distinguish between upper- and lowercase letters, we say that it is CASE INSENSITIVE.

The Unix filesystem is case sensitive. For instance, the letter **a** is completely different from the letter **A**. In the following example, all the file names are considered to be completely different:

```
harley Harley HARLEY haRLey
```

■ **Note** Unless you have a good reason, use only lowercase letters for file names. Some people like to use a single uppercase letter at the beginning of a directory name, but anything more than that is too much.

---

4. DOTFILES

There are many files that, for one reason or another, you will want to ignore most of the time. For example, some programs will put a configuration file in one of your directories. The program will use this file silently, which is fine, but *you* don't want to look at it every time you display the contents of the directory.

By default, the **ls** command will not display the names of files whose names begin with a **.** (dot) character. Such files are called DOTFILES or HIDDEN FILES. If you want to see dotfiles, you use **ls** with the **-a** (all) option. This means that, if you have a lot of configuration files (which are normally dotfiles), you don't have to look at them unless you really want to.

As an experiment, try this. Use **cd** to make sure you are in your home directory. Then use ls to display all the files and directories that are not dotfiles. Then use **ls -a** to display *all* your files, including dotfiles, and notice how many there are.

```
cd
ls
ls -a
```

If you use Bash for your shell, you will see several dotfiles that are the Bash configuration files. For reference, I have listed them in Figure 2-13. I'm showing these to you for two reasons. First, they are a good example of how dotfiles are used. Second, you may want to use Emacs to edit one or more of these files to customize your command line environment. If so, start by looking in the files to see what is already there, figure out what you want, and make your changes carefully.

File	Description
.bash_profile	Login file
.bash_login	Login file
.profile	Login file (POSIX mode)
.bashrc	Environment file
.bash_logout	Logout file

FIGURE 2-13. Bash Configuration Files.

Many programs use dotfiles to hold configuration information. These are the files used by Bash, the default shell on many Unix systems (including most types of Linux as well as Mac OS X). Once you learn how to use Emacs, I suggest that you take some time to learn about these files and customize them.

Note

When you are using the command line, you see dotfiles by using the **ls -a** command.

With a GUI-based file manager, if you want to see dotfiles, you will have to turn on a specific option. For example, with Nautilus (the file manager used with Ubuntu Linux),[32] if you want to see dotfiles, you need to pull down the View menu and select "Show hidden Files". (The shortcut key is <Ctrl-H>.)

Section 2.19: File and Directory Names: OS X and Windows

What we have discussed in Section 2.18 applies to virtually all types of Unix. To finish our discussion, I'd like to take a moment to tell you the differences you will find with Mac OS X and with Windows.

MAC OS X:

Unlike other types of Unix, the OS X filesystem is case insensitive. In other words, it does not distinguish between upper- and lowercase letters. If you are used to Unix, this will be strange to you. For example, with other types of Unix, if you want to use, say, the **ls** command, you must spell it exactly. With OS X, you can use **ls** or **LS** or **lS** or **Ls**.

Similarly, with other types of Unix, you must spell file names exactly. For example, the file **harley** is different than the file **Harley**. With OS X, they are considered to be the same. Thus, with OS X, the following commands are all equivalent:

```
ls harley
LS HARLEY
Ls Harley
lS harLEY
```

[32] In September, 2012, Nautilus was renamed "Gnome Files". However, this is such a boring name, most people still call the program "Nautilus".

However, and this is important, when you type options with OS X, they *are* case sensitive. So, with OS X, the following commands are different:

```
ls -a harley
ls -A harley
```

Another difference between OS X and other types of Unix is that the Mac culture encourages the use of names that include spaces. This means that when you use the command line, you are likely to encounter file and directory (folder) names that have spaces. If so, when you need to reference them in a command, remember to put the entire name in single quotes. For example:

```
cd 'New Programs'
```

MICROSOFT WINDOWS:

Windows, like OS X, is also case insensitive. When you type commands and filenames, you can use either upper- or lower case. The same goes for options (which have a different format than Unix options.)

When you are using commands and you encounter a file with a name that contains spaces, you must put quotes around the file name. However, unlike Unix, you must use double quotes, not single quotes. For example:

```
dir "Saved Games"
```

This Windows command lists the contents of a directory named **`Saved Games`**.

The last important consideration has to do with the filesystem itself. As we discussed in Section 2.16, Unix has one large filesystem, starting from a single root directory. Windows uses a different filesystem for each device. For example, if you have two hard disks, a CD drive, and a memory stick, each device will have its own filesystem with its own root directory.

CHAPTER 3

Installing Emacs

Section 3.1: Installing Software: Packages vs. Manual Installation

In this chapter, I will show you how to install Emacs with Linux, OS X, and Microsoft Windows. Before we get to the details, however, I want to discuss the various ways in which Emacs, and software in general, can be installed. Even if you already have Emacs on your computer, I would like you to take a few minutes and read this entire chapter (which is short). There are two reasons.

First, I am going to explain important concepts that I want you to understand. This is because what we will be talking about is generally applicable to installing all types of software, not just Emacs. Second, regardless of which operating system you are using, I think you will find it interesting to see how software installation differs with Unix and Linux, OS X, and Windows.

Emacs is widely available for free: all you have to do is find it and install it. There are a number of different versions of Emacs but, by far, the most widely used is GNU Emacs (see Section 1.4), and that is the version of Emacs we will be discussing in this book. However, because basic Emacs doesn't vary much, if you are using another type of Emacs, what you learn in this book will work just fine on your system.

Emacs has been adapted to run on a very large number of operating systems. My conservative estimate is that, over the years, various versions of GNU Emacs have run on well over 100 different types of operating systems. (And when I quote you that number I am counting *all* the different Linux distributions as one.) As I write this (2016), the most important operating systems on which GNU Emacs is supported officially are:

- Linux (all distributions)
- FreeBSD
- NetBSD
- OpenBSD
- OS X on the Mac
- Solaris
- Microsoft Windows

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_3

In most cases, you will find that installing GNU Emacs is fast and easy. However, the installation details depend on what operating system you are using. There are two basic ways to install Emacs: using a package management system, or doing a manual installation. I'd like to take a minute to discuss each of these methods briefly, so you understand how they work. Before we start, I'd like to define a few technical terms.

An ARCHIVE is a file that acts as a container for multiple files. The data within an archive is usually compressed to save storage space. When you process an archive in order to access the contents, we say that you EXTRACT the files.

Using an archive is an easy way to store and share any type of file. For instance, you might use an archive to store the full collection of all the photos you have ever taken of your cat. You could then send a copy of the archive — one single file — to your friends. Once they receive the archive, it is easy for them to extract all 739 separate cat photos to enjoy on their own computer.

The most common use of archives is software distribution. In such cases, an archive contains all the programs, data, and metadata (see below) that someone will need to install a specific program on their computer. For example, the Free Software Foundation (FSF) distributes Emacs by using archives. All you need to do is download the appropriate archive for your particular system. You can then extract the files and install them on your system (as long as you know what you are doing).

METADATA refers to information about the contents of the archive. In a software archive, metadata generally consists of helpful comments to be read by humans, as well as directory information, time stamps, file permissions, and file ownership data. You may also see information related to error detection/correction and encryption.

Generally speaking, the programs within a software archive will be either source files or binary files. SOURCE FILES contain programs that need to be COMPILED (processed) before they can be installed and used. BINARY FILES are compiled programs that are ready to run. With Unix (and Linux) systems, programs are generally distributed as source files, so they can be compiled to run on a specific system at the time of installation. Windows programs, on the other hand, can run on other Windows systems, so they are distributed as binary files, which makes the installation quicker and simpler.

With Unix, archives are most commonly created as TARBALLS. The name indicates that the archive is created by the Unix **`tar`** program. Although you will see variations, a typical tarball will have a file name that ends in **`.tar.gz`**. For example, the tarball that contains the archive for GNU Emacs version 24.5 is called **`emacs-24.5.tar.gz`**.

In this example, the **`tar`** program was used to put all the files needed to install GNU Emacs version 24.5 into a container file named **`emacs-24.5.tar`**. This is called a TARFILE. The tarfile was then compressed by the **`gzip`** data compression program to create the tarball named **`emacs-24.5.tar.gz`**.

With Windows, software archives are most commonly shared as ZIP FILES. Conceptually, a zip file is similar to a tarball, in that it is a compressed file containing a collection of other files. However, the underlying technology is different. Moreover, as I mentioned, when archives are used to distribute software, you will find that Windows zip files contain binaries, where Unix tarfiles generally contain source files.

A moment ago, I told you that an archive contains everything you need to install specific software *as long as you know what you are doing*. That is a big "if", because installing software manually on a Unix system requires knowledge and experience.

To make software installation faster and easier, virtually all Unix systems come with a PACKAGE MANAGEMENT SYSTEM or PACKAGE MANAGER to do the work. A package manager automates the various processes required to install, upgrade, configure, and uninstall software. All you have to do is tell the package manager what you want, and everything is done automatically.[1] As a general rule, each Linux distribution uses a specific package manager. By far, the most popular ones are APT and RPM. However, you will also see Pacman, Pgktool and Portage. The various BSD versions of Unix use either **`pkg`** or **`pkg_add`**.

A package manager installs software by using a PACKAGE: a file containing an archive, as well as extra information needed to guide the installation process. There are tens of thousands of different packages, which means it is easy to download and install tens of thousands of different programs. Most packages are shared online in public REPOSITORIES, which are free for anyone to use. (Please take a moment to reflect on this. In fact, the invention of public repositories was one of the main reasons Linux became so important and so popular.)

Each package manager uses its own type of packages. For example, if you want to install Emacs on a Linux system that uses APT, you need an APT Emacs package; if you have a system that uses RPM, you need an RPM Emacs package. However, this isn't something you need to worry about. All you need to do is tell your package manager what you want, and it will search the appropriate online repositories, find the package you want, download it, and install the software for you. Everything is automatic — and free.

If, for some reason, the program you want is not available as a package but you can find it in the form of an archive — a tarball for Unix or a zip file for Windows — you can download it yourself and do a manual installation. I'll show you how to do this for Emacs in Section 3.3 (Linux) and Section 3.5 (Windows).

Finally, installing Emacs with OS X uses a completely different procedure, which I will show you in Section 3.4.

[1] See Personal Note #27, "Comparing Unix Packages to Commercial Apps" (Appendix A).

Section 3.2: Installing Emacs Using a Linux Package Manager

Each type of Linux uses a specific package management system (package manager). There are a variety of Linux package managers, the most widely used being APT and RPM. For reference, Figure 3-1 shows you the Linux package managers you are most likely to encounter. As you can see, APT is used by all Debian-based Linux distributions,[2] and RPM is used by Fedora-based and SUSE-based distributions. Figure 3-2 shows you the package managers used by the three most popular BSD operating systems.

Package Manager	Linux Family	Linux Distributions
APT	Debian-based	Debian, Mint, Ubuntu
RPM	Fedora-based	Fedora, Mageia, Manjaro, RHEL, CentOS
RPM	SUSE-based	OpenSUSE
Pacman	Arch-based	Arch
Portage	Gentoo-based	Gentoo
`pgktool`	Slackware-based	Slackware

FIGURE 3-1. Linux Package Management Systems.

To make software easy to share and download, most Unix systems use a package management system. The most widely used Linux package managers are shown in this table.

Package Manager	BSD System
`pkg`	FreeBSD
`pkg_add`	NetBSD
`pkg_add`	OpenBSD

FIGURE 3-2. BSD Package Management Systems.

This table shows the package managers you are most likely to encounter when you use a BSD-based version of Unix.

The package managers I have mentioned are all CLI-based programs. That is, you use them by typing commands. For completeness, I will mention that some types of Linux also have GUI-based programs. (For example, Ubuntu has a GUI-based package manager named Synaptic.) However, I won't be discussing them here because the CLI-package managers are more widely used and, in my opinion, are more fun to use.

[2] Debian Linux was created in 1993 and is still actively developed. Over the years, Debian has spawned a large number of derivative Linux distributions, including the popular Ubuntu and Mint families.

In this section, I will show you how to install Emacs using APT and RPM. If your operating system uses a different package manager, you can look it up online to see how it works, or you can use the information in Section 3.3 to install Emacs manually.

Installing Emacs using a package manager is easy. All you have to do is enter the appropriate command and everything will be done for you automatically.

Installing Emacs with APT

To install software with the APT package manager, you use the **`apt-get`** program. To install Emacs, the command to use is:

```
sudo apt-get -y install emacs
```

Let's discuss this command, one piece at a time:

- **`sudo`** runs the command as superuser, so you will have permission to install software on the system. You will need the superuser/administrator password to run this command. (We discussed the superuser in Section 2.3.)
- **`apt-get`** is the program you want to run.
- The **`-y`** option: During the installation process, **`apt-get`** will pause and ask your permission to proceed with various tasks. The **`-y`** option automatically answers "yes" to such prompts, so **`apt-get`** will not pause. If you so desire, you can leave out this option and respond to the prompts manually.
- **`install`** tells **`apt-get`** that you want to install software.
- **`emacs`** is the name of the package you want to install.

Once you have installed Emacs, you can test it by starting the program:

```
emacs
```

To quit Emacs, press <Ctrl-C> <Ctrl-X> .

To uninstall Emacs, use the following command:

```
sudo apt-get -y remove emacs
```

Using **`remove`** deletes the package but leaves its configuration files, which were used to help install the software.

By default, these files are left alone in case you have modified them. However, if you do not plan to reinstall the program or install a new version of the program, you may want to delete both the package and its configuration files. To do so, use:

```
sudo apt-get -y remove --purge emacs
```

Installing Emacs with RPM

To install software with the RPM package manager, there are two programs you might use. The best choice is DNF. To install Emacs, the command to use is:

```
sudo dnf -y install emacs
```

To uninstall Emacs, use the following command:

```
sudo dnf -y erase[3] emacs
```

DNF is a replacement for an older program called YUM. DNF is better than YUM, so it should be your first choice.[4] However, if your system uses RPM but it doesn't have DNF, you can use YUM. The syntax is the same as DNF:

```
sudo yum -y install emacs
udo yum -y erase emacs
```

Once you have installed Emacs, you can test it by starting the program:

```
emacs
```

To quit Emacs, press <Ctrl-C> <Ctrl-X> .

Section 3.3: Installing Emacs Manually With Linux

The following instructions show you how to install GNU Emacs manually on a Debian-based Linux system (this includes the Ubuntu and Mint families). A manual installation involves downloading a tarball from the GNU Emacs archive, and then unpacking, compiling, and installing the program. Normally, you would use a package manager, which is a lot easier. However, in some cases, you will want to do a manual installation. Here is an example.

When I was working on this book, the most up-to-date version of Emacs was 24.5. At the time, the latest version of Emacs for which there was a package in the Debian repository was 24.4. (It can take a while for a package to be created when a new version is released.) I could use **`apt-get`** to install Emacs 24.4 but, for technical reasons, I wanted to test 24.5, so I had to install it manually.

Below I will show you the procedure I followed. Although the details (such as version numbers) will change as time passes, the general principles won't, so you can modify the following commands to suit your needs.

---

[3] You can use either **`erase`** or **`delete`**, but **`delete`** is deprecated.

[4] For example, DNF is much better at handling dependencies.

When you read my annotated procedure for installing Emacs by hand, do take a moment to make sure you understand each step. At the end of the section, I will give you all the commands in a quick list, to make it easy for you to enter them one after another. So don't do anything right now. Just read through my comments until you get to the list of commands at the end of the section.

1. Before you start, be sure to uninstall any existing Emacs programs that may already be on your system. For example, if you have previously installed Emacs using **`apt-get`** you can use the command:

```
sudo apt-get -y remove emacs
```

2. On Debian systems, the tools you need to build a program from source code are not installed by default. So before you can compile and install anything, you need to install these tools (compilers, libraries, and so on). They are in a package called **`build-essential`**. The command to use is:

```
sudo apt-get install build-essential
```

IMPORTANT: Be sure to type **`essential`**, not **`essentials`**.

3. Make sure your system satisfies the build dependencies for Emacs version 24:

```
sudo apt-get build-dep emacs24
```

IMPORTANT: Be sure to specify only the major version number. Do not include the minor number. In our example, I used **`emacs24`**, not **`emacs24.5`**. If you include the minor number (**`24.5`**) it won't work, and it will take a long time to figure out why.

Note

When you run the **`apt-get build-dep`** command for the first time, you may reach a point where the program pauses to ask you to specify the "Postfix Configuration". (Postfix is a mail server, a program that delivers email over a network. Configuring Postfix is important if you plan to send email from Emacs.)

You will see a text menu with several choices. Use the Tab key to move to the configuration type that best describes your system. Then press <Enter> to make your selection. If you are not sure what to do, select "No configuration".

---

3. Change to your home directory and create a subdirectory to hold the installation files:

```
cd && mkdir emacs-24.5-install
```

4. Change to the installation directory:

```
cd emacs-24.5-install
```

5. Download the tarball you need to install Emacs. First, use your Web browser to visit the GNU Emacs FTP server:

```
http://ftp.gnu.org/gnu/emacs/
```

Look for the appropriate tarball, and download it using **wget** (a program to retrieve content from Web servers):

```
wget http://ftp.gnu.org/gnu/emacs/emacs-24.5.tar.gz
```

The file will be downloaded into the current directory (in this case, **emacs-24.5-install**).

6. Extract the installation files from the tarball.

```
tar -xvzf emacs-24.5.tar.gz
```

At this point, the Emacs source programs will be in a subdirectory with the name taken from the tarball: in our case, **emacs-24.5**.

7. Change to the subdirectory that holds the installation files:

```
cd emacs-24.5
```

8. Configure the Emacs software:

```
./configure
```

9. Build the Emacs software:

```
make
```

10. Install Emacs:

```
sudo make install
```

11. Test to see if Emacs is installed properly, by starting the program:

```
emacs
```

To quit Emacs, press <Ctrl-C> <Ctrl-X>.

For reference, here are all the commands in an easy-to-read list. Remember, before you start, make sure that you have uninstalled any existing version of Emacs:

```
sudo apt-get -y remove emacs
```

Also, visit the GNU Emacs FTP server and find the tarball you want:

```
http://ftp.gnu.org/gnu/emacs/
```

In our example, it is **emacs-24.5.tar.gz**. In the commands below, change the version numbers appropriately.

Here are the commands you need to install Emacs manually on a Debian-based Linux system:

```
sudo apt-get install build-essential
sudo apt-get build-dep emacs24
cd && mkdir emacs-24.5-install
cd emacs-24.5-install
wget http://ftp.gnu.org/gnu/emacs/emacs-24.5.tar.gz
tar -xvzf emacs-24.5.tar.gz
cd emacs-24.5
./configure
make
sudo make install
```

To see if Emacs is installed properly, enter:

```
emacs
```

To quit Emacs, press <Ctrl-C> <Ctrl-X> .

Section 3.4: Installing Emacs With OS X

Installing Emacs with OS X is a multi-step process. First you modify your system settings to allow you to run a program downloaded from the Internet. Then you download and mount a disc image of the Emacs installation software. You then install Emacs, and create an icon for it in the Dock. Finally, you start Emacs to make sure it works.

1. ENABLE THE RUNNING OF PROGRAMS DOWNLOADED FROM THE INTERNET

Before you can install Emacs with OS X, you need to make sure your settings allow you to run programs downloaded from the Internet.

a. Open Finder, click "Applications", then double-click "System Preferences". The "System Preferences" will open.

b. Within "System Preferences", click "Security & Privacy". The "Security & Privacy" window will open.

At the bottom of the General Tab where it says "Allow apps downloaded from:"
If "Anywhere" is selected, that is fine. Close the window.
If not, you need to change the setting to "Anywhere".

c. Click the icon of a lock next to "Click the lock to make changes". You will be asked to type your password. Type the password you use to log in to your account, then click "Unlock".

d. At the bottom of the General Tab where it says "Allow apps downloaded from:", select "Anywhere".

e. If you see a warning message, read it and then click "Allow From Anywhere". If you see a window with a warning that this setting will change back after 30 days, click "Allow From Anywhere" to approve the change.

f. To finish, lock the setting by clicking the icon of the lock next to "Click the lock to prevent further changes".

g. Close the "Security & Privacy" window.

2. DOWNLOAD A DISK IMAGE OF THE INSTALLATION SOFTWARE

To install Emacs, you must download a DMG ("disk image") file. A DMG file holds the image of an optical disc, so you can think of it as a virtual disc. (A DMG file is the Mac version of an ISO file.)

The Web page to download a DMG file containing Emacs for OS X is:

`http://www.emacsformacosx.com/`[5]

Start your browser, visit this page, and download the DMG file. The file should end up in your Downloads folder. If you can't find it there, check your desktop. Once you are sure the file is downloaded, you can close your browser.

3. MOUNT THE DMG FILE

When you MOUNT a disc image file, it emulates your putting a real disc into your CD or DVD device. If you have any experience using ISO files with Unix or Windows, you will see that mounting a DMG file on a Macintosh is like mounting an ISO file, only easier.

All you need to do is use Finder to open the Downloads folder. Then double-click on the DMG file and wait. Your disc image will be mounted automatically. As this happens, you will see a notice that the DMG file is being opened. Once the DMG file is mounted, you will see an icon for the program (in this case, an Emacs icon).

4. INSTALL EMACS

Drag the Emacs icon to your Applications folder. It is that easy.

5. CREATE A DOCK ICON

Use Finder to open the Applications folder. Find the Emacs icon, and drag it to the Dock for easy access. You can now close the Applications folder. (The Dock is like the Ubuntu Launcher or the Windows Taskbar.)

---

[5] If this Web site doesn't work, search online for: **`installing emacs "os x"`**.

6. START EMACS TO MAKE SURE IT WORKS

Click the Emacs icon in the Dock to start the program. If your permissions have not been set up properly, you will see:

"Emacs" can't be opened because it was not downloaded from the Mac APP store

If so, simply follow the instructions in Step 1 above. Otherwise, the first time you run Emacs, you will see:

"Emacs" is an application downloaded from the Internet. Are you sure you want to open it?

Click "Open" to start the program. If necessary, bring the focus to the Emacs window by clicking on it.

7. QUIT EMACS

To quit Emacs, press: <Control-X> <Control-C>. (Be sure to use <Control>, not <Command>.)

8. "EJECT" THE DISC IMAGE

Now that you are finished with the virtual disc, you can "eject" it. Look on your Desktop for a white and gray icon with the name Emacs. This represents the mounted DMG file.

Click on the icon. Then press <Command-E>, which will instantly "eject" the virtual disc, at which time the icon for the mounted DMG file will disappear.

9. DELETE THE DMG FILE

Use Finder to open the Downloads folder. Drag the DMG file to the Trash folder. If you want, empty the Trash. Close Finder.

UNINSTALLING EMACS

If you ever want to uninstall Emacs, the process is easy: just drag the Emacs icon from the Applications folder to the Trash.

To remove the Emacs icon from the Dock, right-click on the Emacs icon and select Options, and then "Remove from Dock".

■ **Note**

When you use a mouse, there are two basic ways to click: a regular "click" and a "right-click". On a Macintosh, these two types of clicks are referred to formally as a PRIMARY CLICK (click) and a SECONDARY CLICK (right-click). When I instructed you (above) to right-click on the Emacs icon, I was referring to a secondary click.

With a Macintosh two-button mouse, one button sends the primary click; the other sends a secondary click. If you have a one-button mouse, to create a secondary click (a right-click), hold down the Control key as you click.

With a Macintosh touchpad, one of the following should work:

- Touch the touchpad with two fingers.
- Touch the bottom right corner of the touchpad.
- Touch the bottom left corner of the touchpad.

It is a good idea to set the primary and secondary clicks on your system to suit your preferences. To do so, use Finder to open the Applications folder. Then open "System Preferences", and choose either Mouse or Trackpad.

Section 3.5: Installing Emacs With Microsoft Windows

To install Emacs with Microsoft Windows you don't run a setup program, as is usually the case with Windows software. Instead, you download the appropriate archive and then extract its contents. You can then pin the Emacs program to your Taskbar or create an icon for it.

Before you start, you need to choose a directory into which you will install Emacs. If you don't have a master plan for your directory structure, I suggest you use **`C:\Emacs`**, which is what I will use in the examples below.

■ **Note** During this procedure, you will need to use the Windows file manager. In Windows 7, the file manager is Windows Explorer. In Windows 8 and Windows 10, it is File Explorer.

---

1. CREATE AN EMACS DIRECTORY

Use the Windows file manager to create the directory **`C:\Emacs`**.

2. DOWNLOAD THE EMACS ARCHIVE

Use your Web browser to visit the GNU Emacs FTP server where the Windows archives are stored:

`http://ftp.gnu.org/gnu/emacs/windows/`

Look for the version of Emacs you want to install. (I suggest the latest version.) In our example, I will use:

`emacs-24.5-bin-i686-mingw32.zip`

Click on this link to download the file to your computer. Be sure to save the file to the **`C:\Emacs`** directory (not to your Downloads directory).

Note

When you download an archive (or a setup program) for Microsoft Windows, it will contain binary files, all ready to run. This is convenient, but it also makes it easier to infect your computer with malware, such as viruses or trojans. With binary files, there is no way of knowing what's inside the installation package. With source files, anyone can look inside the programs, so the programs are open to the world.

For this reason, before you install any Windows program, it behooves you to use an up-to-date antivirus program to check the installation file for malware.

---

3. EXTRACT THE CONTENTS OF THE ARCHIVE

Use the Windows file manager to navigate to **`C:\Emacs`**. Look for the archive (zip file) you just downloaded. To extract its contents, right-click on the file name and choose "Extract All".

You will see a dialogue asking you to select the directory to hold the contents of the archive. Before you proceed, make sure that there is a checkmark next to "Show extracted files when complete".

You will see a suggestion to create a directory with the same name as the first part of the zip file. In our example, it is:

`C:\Emacs\emacs-24.5-bin-i686-mingw32`

Unless you have a good reason to use another directory, accept the default choice. To process the archive, click Extract.

Once the extraction process is finished, you will see a file manager window showing you the contents of the destination directory. Specifically, you should see four directories: **`bin`**, **`libexec`**, **`share`** and **`var`**. Within these directories are some subdirectories and a large number of files.

At this point, your Emacs installation is complete.

■ **Note**

In a moment, you will be running Emacs. The first time you run the program, you may see a warning:

```
The publisher could not be verified.
Are you sure you want to run this software?
```

As long as you downloaded the Emacs archive from the official GNU Emacs FTP site, it is safe to uncheck the box next to "Always ask before opening this file". That way, you won't be bothered every time you want to run the program.

3. START EMACS TO MAKE SURE IT WORKS

Within the file manager, change to the **bin** directory. In our example:

```
C:\Emacs\emacs-24.5-bin-i686-mingw32\bin
```

You will see a number of executable programs (**.exe** files). To start Emacs, run the program **runemacs.exe**. Once Emacs is running, you can quit it by pressing <Ctrl-X> <Ctrl-C> .

4. MAKE EMACS EASY TO RUN

To make Emacs easy to run whenever you want, you can pin it to the Taskbar. To do so, use the file manager to navigate to the Emacs bin directory. Right-click on **runemacs.exe** and select "Pin to Taskbar".

To create an Emacs icon, run the program **addpm.**exe. (It's in the same directory.)

5. CLEAN UP

You can now delete the Emacs archive file. In our example, it is:

```
emacs-24.5-bin-i686-mingw32.zip
```

CHAPTER 4

The Emacs Keyboard

Section 4.1: A Strategy for Learning Emacs

In the first three chapters of this book, we discussed:

- What is Emacs, and where did it come from?
- Basic Unix skills and how they relate to Emacs.
- Installing Emacs on your own computer.

At this point, you are now ready to begin learning how to use Emacs.

With most text editors, the way to start is to learn some of the basic keystrokes — how to move the cursor, how to page up and down, how to search for a pattern, and so on — and then practice, practice, practice.

With Emacs you need a different strategy. As we discussed in Section 1.3, Emacs is wonderful in that it is a full-fledged working environment. However, it is this very same exhaustive complexity that makes Emacs difficult to learn. So, here then, are three helpful guidelines. First, what *not* to do:

1. Do not jump in and start by learning the basic keystrokes. Keystrokes are easy to learn. To really understand Emacs, you must first have the proper background (Chapters 1 and 2 of this book).

2. As you will see in Section 12.4, Emacs comes with a built-in tutorial. Do not begin by starting Emacs and firing up the tutorial. All that will happen is you will become confused and discouraged. (At least, that's what happened to me.)

So what should you do?

3. After you have read Chapters 1 and 2, continue by reading each section of this chapter in order. I will start by teaching you all the basic concepts (and there are a lot of them). At the proper time, I will show you how to use the fundamental keystrokes, and *then* you can practice, practice, practice. When you get to Section 12.4, I will explain how to run the Emacs tutorial, and you can use it as a post-graduate course.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_4

Section 4.2: The Ctrl Key

Emacs has a lot of key combinations, referred to as "key sequences". (We'll talk about key sequences Section 6.1, at which time I will give you a technical definition.) There are far more key sequences than you will ever memorize. In Section 4.5, I will explain why there are so many. First, though, you need to understand how Emacs uses the keyboard.

Like all text editors, Emacs uses all the regular keys (letters of the alphabet, numbers, punctuation, and so on). However, Emacs also uses two special keys: Ctrl and Meta.

The Ctrl key is used in the usual way. You hold down <Ctrl> as you press another key. For example, the command <Ctrl-H> starts the built-in Help facility: hold down the Ctrl key and press the letter **h**. No surprise here.

What will be new to you is that many Emacs commands consist of more than one Ctrl combination in a row, or a Ctrl combination followed by a single letter. Here are two examples:

- To quit Emacs, you use <Ctrl-X> <Ctrl-C>. That is, press <Ctrl-X> and then press <Ctrl-C>.
- To start the built-in tutorial, you use <Ctrl-H> **t**. That is, you press <Ctrl-H> and then press the letter **t**.

In order to make the description of such key sequences readable, Emacs uses its own notation. As you may know, the Unix convention for describing the Ctrl key sequences is to use a **^** (circumflex) character to represent <Ctrl>. For example, in the Unix world, **^X** means <Ctrl-X>. Another part of this convention is that when we describe a Ctrl key sequence, we write the letter of the alphabet in uppercase. For instance, we write **^X**, not **^x**. We do this because it is easier to read. (You don't actually press the Shift key when you type the **x**.)

In Emacs, the Ctrl key is represented by **C-** (the uppercase letter "C" followed by a hyphen), and the letters are written in lowercase. For example, instead of writing <Ctrl-X> or **^X**, we write **C-x**. Similarly, the combination <Ctrl-X> <Ctrl-C> is written as **C-x C-c**; and <Ctrl-H> followed by the letter **t** is written as **C-h t**. It is important that you recognize this notation because that is what you will see when you read about Emacs. To help you get used to these conventions, I will use them consistently throughout the rest of the book.

Section 4.3: The Meta (Alt) Key

In addition to the Ctrl key, Emacs uses the Alt key which, for historical reasons, is referred to as the META KEY, written as <Meta>, so whenever you see a reference to the Meta key, just press <Alt>. (With a Macintosh keyboard, the Meta key is the Option key.)

The Meta key is used in the same way you use the Ctrl and Shift keys. That is, you hold it down while you press another key. For example, to type an uppercase "A", you hold down the Shift key and press the letter **a**. To type <Ctrl-A>, you hold down <Ctrl> and press **a**. And, to use <Meta-A>, you hold down <Alt> and press **a**. (Or, with a Macintosh keyboard, you hold down <Option> and press **a**.)

The Emacs notation used to indicate a Meta key sequence is similar to what we use with the Ctrl key, except that we use **M-** instead of **C-**. For instance, to indicate the combination <Meta-A>, we write **M-a**. Here is an example:

- When you are editing a file, the command to move down one screenful is **C-v**. The command to move up one screenful is **M-v**.

This means, to move down one screenful, you press <Ctrl-V>. To move up one screenful, you press <Meta-V> (that is, <Alt-V>).

As an alternative, you can also use the Escape key (<Esc>) as a Meta key. However, when you use <Esc>, you do not hold it down. You press it, let go, and then press the second key. For example, if you want use the **M-v** command I mentioned above, you can either press <Meta-V> or type the two keys <Esc> **v**. (This alternative way of using <Meta> was created because, at one time, there were keyboards that didn't have a Meta key or an Alt key.)

You will occasionally see key sequences that use both the Meta key and the Ctrl key. For example, **M-C-s**. To type this command, you have two choices. You can either hold down <Ctrl> and <Meta> and press **s**, or you type the two keys <Esc> <Ctrl-S> .

Thus, there are four possible ways to use each letter of the alphabet. For instance, the letter "a" can be used as a:

- Lowercase letter (**a**)
- Uppercase letter (**A**)
- Ctrl combination (**C-a**)
- Meta combination (**M-a**)
- Meta-Ctrl combination (**M-C-a**)

When you have a Meta-Ctrl combination, you hold down both keys at the same time. For example, the following two combinations are equivalent, and you will see it written both ways.

```
M-C-a
C-M-a
```

I know this all may sound confusing when you read it, but in practice it is easy.

Section 4.4: Special Key Names

When you read about Emacs, you will see special names used to represent specific keys. These names are shown in Figure 4-1. It is important that you memorize them, as you will see these names when you read Emacs documentation, especially reference information about commands and Lisp functions.

Name	Key
C-	Ctrl
M-	Meta
M-C-	Meta-Ctrl
BS	Backspace
DEL	Delete
ESC	Esc
RET	Return
SPC	Space
TAB	Tab

FIGURE 4-1. Emacs names for special keys.

When you read about Emacs, you will sometimes see abbreviated names used for special keys. These names are derived from the technology of the 1970s, and they are important enough that it is a good idea to memorize them.

What you see in Figure 4-1 comes from very old technology of the 1970s, when Unix and Emacs were originally developed. Except for the Meta key (which we'll talk about in a moment) and the Tab key, all the keys in Figure 4-1 can be traced back to the earliest Unix terminal, the Teletype Model 33 ASR we discussed in Section 2.4.

We have already met the Ctrl key and Meta key. To summarize:

- **C-** stands for "hold down the Ctrl key"
- **M-** stands for "hold down the Meta key"
- **M-C-** stands for "hold down both the Meta key and the Ctrl key"
- As an alternative to the Meta key, you can press <Esc>.

For example, **M-x** means you can use either <Meta-X> or <Esc> **x**. **M-C-x** means you can use either <Meta-Ctrl-X> or <Esc> <Ctrl-X>.

You will see the Meta-Ctrl double-key sequences written in three different ways, but they all mean the same thing. For example, the three commands below are equivalent. They mean either "hold down the Meta and Ctrl keys and press **s**, or" press the Esc key, let it go, and then press <Ctrl-S>.

C-M-s
M-C-s
ESC C-s

Here is another, more complex example. While I was researching this chapter, I found a reference that contained a long list of Emacs commands. It the middle of the list was a line that said:

```
ESC-!    Enter a shell command.
```

In other words, while you are working within Emacs, you can enter a shell command by typing **ESC !** (followed by the shell command). The moment I saw this, my mind changed **ESC-!** to <Meta-!>. (This is the type of thinking I want you to train you mind to do automatically.)

But there is more. Look at your keyboard and notice that the **!** (exclamation mark) character is actually an "uppercase" **1** (the number 1). Thus, the sequence **ESC !** is actually **ESC** <Shift-1>, which is the same as <Meta-Shift-1>, which is actually <Alt-Shift-1> on my keyboard.

So when I read that the way to enter a shell command is by using **ESC !**, my mind immediately knew to press <Alt-Shift-1>, type the shell command, and then press <Enter>. One day, your mind will do the same and, on that day, you will know you are an Emacs person.

To continue with the discussion of Figure 4-1. The **BS**, **DEL**, **ESC**, **RET**, **SPC**, and **TAB** keys are simple to understand: they represent the following keys: Backspace, Delete, Esc (Escape), Return (Enter), Space, and Tab. (**DEL** refers to the same Delete key we discussed in Section 2.10.)

Note

On a Macintosh keyboard, BS refers to the Delete key at the top-right corner of the main part of the keyboard. This is the primary Delete key, which corresponds to the Backspace key on a PC keyboard. DEL refers the secondary Delete key, the one to the left of the End key.

The secondary Delete key is available only with a full-sized Macintosh keyboard. If you have a compact keyboard, you won't have this key. Instead, you can use either <fn-Delete> or <Control-D>.

Section 4.5: The Meta Key, Bucky Bits, and Much More

As soon as you start to use Emacs, you will find out how important the Meta key is. In fact, the use of the Meta key is one of the defining characteristics of Emacs. And yet, I hear you say, "What Meta key? Ain't no stinkin' Meta key on my keyboard." Indeed, modern keyboards don't have a Meta key, we use <Alt> instead (or <Option> with a Macintosh). So why don't we just call it the Alt key?

The Meta key is a legacy from the early days of Emacs and Lisp at MIT (see Section 1.4), when there actually were keyboards that had this key. In fact, there were at least two such keyboards that were used with Lisp machines. (Remember, Emacs is tightly integrated with the Lisp programming language.) These keyboards were called the Knight keyboard and the Space Cadet keyboard. I'll talk more about them in a moment but, first, I want to explain a few technical details.

In Section 1.2 we talked about ASCII (the American Standard Code for Information Interchange). The list of all the ASCII characters and their meanings is called the ASCII CODE. The ASCII code uses 8-bit bytes to store characters, one character per byte. The original, plain vanilla ASCII code used only 7 of the 8 bits and, thus, defined 128 characters from 0 to 127. The original purpose of the Meta key was to turn on the top (8th) bit to allow the use of characters 128 through 255. In other words, using a Meta key effectively doubled the number of different characters from 128 to 256.

The tradition of using a special key to modify the top bits of a character began with the Stanford University SAIL KEYBOARD, developed in 1971 for special-purpose Lisp computers at the Stanford Artificial Intelligence Lab (SAIL). These keyboards had a number of extra keys, including Meta, Control and Alt.

Shortly afterwards, the West Coast SAIL keyboard began to influence the design of keyboards used by the East Coast AI (artificial intelligence) community. First, the so-called KNIGHT KEYBOARD was created by Tom Knight to be used on the legendary MIT Lisp Machines at MIT's AI Lab, where Emacs was later developed. (A LISP MACHINE was a single-user computer, optimized to run the Lisp programming language.) This keyboard and the Lisp working environment heavily influenced Richard Stallman when he designed Emacs.

Later, Tom Knight enhanced the Knight keyboard to create the more complex SPACE CADET keyboard (see below). Finally, the Space Cadet keyboard was adapted into several versions of a simpler SYMBOLICS KEYBOARD, used on commercial Lisp machines in the early 1980s.

All of these keyboards had a Meta key, which is how it found its way into Emacs. However, there was more. The Knight keyboard had four modifier keys: Top, Shift, Meta, and Ctrl. The Space Cadet keyboard had seven such keys: Top, Greek, Shift, Hyper, Super, Meta, and Ctrl.

Each key on the Space Cadet keyboard had three markings: in the regular place, higher up (top), and on the front of the key. In the regular place were the standard characters: letters, numbers, and punctuation, exactly like a regular keyboard. Above the regular character were special symbols, mostly mathematical. In fact, the "top" keys on the Space Cadet keyboard contained the complete APL character set. (APL is a mathematical programming language created by Ken Iversen in the 1960s.) On the front of the keys were Greek letters. To access the extra keys, you used <Shift>, <Top>, and <Greek> .

In this way, the Space Cadet keyboard lets you to type four completely different sets of characters. First, you type lowercase characters, numbers and punctuation in the regular way. Second, you hold down the Shift key to type uppercase letters and punctuation, also in the regular way. Third, you hold down the Top key to type the mathematical symbols and the Greek key to type the Greek letters. In other

words, the Shift, Top, and Greek keys worked like regular shift keys. (Note: Because the Greek letters are on the front of the keys, you will sometimes see the Greek key referred to as the Front key.)

The other four modifier keys — Ctrl, Meta, Hyper, Super — used extra bits that were tacked on to the actual characters. For example, in the same way that you can type, say, <Ctrl-A> to send a special signal to the computer, you could also type <Meta-A>, <Hyper-A>, and <Super-A>, all of which were different. (In fact, in is actually possible to still use a Hyper key and a super Key with Emacs. All you have to do is use Emacs Lisp commands to make specific keys on your keyboard become the Hyper and Super keys. You can then connect Emacs commands that don't already have a key sequence — of which there are many — to use whichever Hyper and Super key sequences you want. By the way, if you are using a PC keyboard with Linux, by default, the Windows key is the Super Key.)

To return to the Space Cadet keyboard, in all, you could use it to type more than 8,000 different characters. This phenomenon gave rise to the Emacs (and the general hacker) philosophy that it is worthwhile to memorize the meanings of a very large number of strange, complicated key sequences, if it will reduce typing time.

Note

If you are experienced enough with Unix to have encountered the never-ending debate of Emacs vs. the **`vi`** text editor, here is the gist of it in one sentence:

Emacs people believe that it is worthwhile to memorize the meanings of a very large number of strange, complicated key sequences, if it will reduce typing time.

The extra bits used by the Ctrl, Meta, Hyper, and Super keys are known as BUCKY BITS. They are named after Niklaus Worth, the inventor of the Pascal programming language, whose nickname was Bucky. When Worth was at Stanford in 1964-1965, he suggested adding an extra key (called <Edit>) to use the 8th bit within bytes that stored 7-bit ASCII characters. Although this exact suggestion was never implemented, it did inspire the idea to use extra keys (and extra bits) to extend a keyboard's character set. As the story goes, Worth had prominent front teeth (buck teeth), so behind his back some people called him Bucky. Hence the name bucky bits.

One last point: the Knight keyboard was named after Tom Knight, one of the Lisp Machine's principal designers. However, the name also has more metaphysical connotations in that it recalls a semi-mythical organization of Lisp hackers called the Knights of the Lambda Calculus. (The lambda calculus is the mathematical theory upon which the Lisp programming language is based.)

Section 4.6: Meta Key Problems When Using a Terminal Window

When you run Emacs within a terminal window (see Section 2.6 and Section 5.2), you may encounter a problem using the Meta key.

With most keyboards, the Meta key is the Alt key. However, terminal windows are GUI-based programs that use the Alt key to access top-level menus. For example, it is common to access the File menu by pressing <Alt-F>. This can interfere with using the Meta key, because when you type certain key sequences, the terminal window program will grab the <Alt> key, instead of leaving it for Emacs to interpret. Here is a common example.

With Ubuntu Linux, the default terminal window is a Free Software Foundation program called Gnome Terminal. By default, Gnome Terminal uses <Alt> key combinations to let you access menus. For instance, to access the View menu, you type <Alt-V>. This means that the Emacs command **`M-v`** (which you will meet in Section 7.1 and Section 8.5) won't work properly. All that will happen is you will pull-down the View menu for the terminal window program.

In you have this problem, you need to find an option to disable <Alt> as a menu shortcut key. For example, with Gnome Terminal, pull down the View menu and select "Keyboard Shortcuts". You will see the following option:

`Enable menu access keys (such as Alt+F to open the File menu)`

Once you turn off this option, Gnome Terminal will ignore the Alt key, and your Emacs Meta key will work properly.

Of course, none of this matters when you run Emacs within a virtual terminal (see Section 2.6).

CHAPTER 5

Starting and Stopping Emacs

Section 5.1: Starting Emacs

In Chapter 3, we talked about how to install Emacs on different types of systems. If you are working with a GUI and at the time of installation you created an Emacs icon, all you need to do is click (or double-click) that icon to start Emacs. In this section, I will show you how to start Emacs from the command line. (We discussed how to enter Unix commands in Section 2.9.)

To start Emacs, you enter the Emacs command. The basic syntax is:

`emacs` [*option...*] [*file...*]

where *file...* indicates one or more files you want to edit.

The Emacs command has a lot of options, but you probably won't need them. I'll talk about a few of the options, but if you want to see all of them, use the **`man`** emacs command to look at the Emacs man page. (The online manual is explained in Section 2.13.)

The recommended way to start Emacs is to simply enter the command by itself:

```
emacs
```

Once Emacs starts, you can either create a brand new file or tell Emacs to read an existing file.

When you start Emacs, it automatically displays a screenful of useful information called the SPLASH PAGE. Although this is helpful, after a while, you won't need it. To start Emacs without a splash page, use the **`-Q`** (quick start) option:

```
emacs -Q
```

(Be sure to use an uppercase **`-Q`**. The **`-q`** option is completely different, and is something you probably don't want.)

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_5

■ **Note**

When Emacs starts, it looks for an initialization file named **.emacs** in your home directory. If an initialization file exists, Emacs will read it and execute all the commands it contains as part of the startup process. By placing commands in your **.emacs** file, you can customize your working environment (see Section 11.8).

When you learn more about Emacs, you will want to create an initialization for yourself. However, once you do, you need to stop using the **-Q** option. The reason is that, although -**Q** option tells Emacs not to display a splash screen (which is why we are using it), it also tells Emacs to ignore your initialization file. Thus, if you start Emacs with **-Q,** you won't have your personal customizations.

The solution is to create a **.emacs** file that contains the following command to suppress the splash screen. Once you have this command in your **.emacs** files, you can omit the **-Q** option. (The line that starts with a semicolon is a comment.)

```
; Do not display the splash screen
(setq inhibit-startup-screen t)
```

If none of this makes sense to you right now, don't worry about it.

If you specify the name of a file when you start the program, Emacs will automatically load the file you specified. For example, say that you want to edit an existing file named **document** in the current directory (see Section 2.17). You can enter:

```
emacs document
emacs -Q document
```

Once Emacs has started, you will be ready to edit the file. If you want, you can enter more than one file name. For example:

```
emacs document names addresses phone-numbers
emacs -Q document names addresses phone-numbers
```

Emacs will read each file into its own work area, and you can switch back and forth as necessary.

If you are experimenting as you read, the quick way to stop Emacs is to type **C-x C-c**. We'll talk more about stopping Emacs in Section 5.5.

Note

Although it is possible to specify a file name (or multiple file names) when you start Emacs, this is not the recommended procedure. Why? Because it runs counter to the Emacs philosophy.

Emacs was designed to provide a total working environment. Unlike other editors, Emacs makes it easy to handle more than one file at the same time (as well as use email, work with directories, write and debug programs, enter shell commands, and so on).

The intention is that you should start Emacs by entering the command without file names, and then initiate as many different tasks as you want, switching from one to another as the mood takes you. Once you get good at Emacs, you can do just about anything you want from within it. Indeed, some people virtually live within Emacs (leaving only to get something to eat).

So, although you *can* specify a file name when you start Emacs, in the long run, you are better off thinking of Emacs as home away from home and not simply a text editor.

Section 5.2: Starting Emacs in a Terminal Window

When you start Emacs from a terminal window, you are working within the GUI. (For a discussion of terminal windows compared to virtual terminals, see Section 2.6.) Emacs knows when you start it within a GUI and, by default, it will use a GUI-based format. For example, if you are in a terminal window and you enter the following command, you will get the GUI version of Emacs:

```
emacs
```

If you prefer using Emacs like this, and you don't want to tie up your terminal window, you can tell the shell to run the program in its own window, that is, as a separate process. To do so, all you need to do is append an **&** (ampersand) character to the end of the command. Here are some examples:

```
emacs &
emacs document &
emacs document names addresses phone-numbers &
```

When you enter the command in this way, the GUI version of Emacs runs in its own window, and your terminal window is still available for other work.

Alternatively, if you prefer to use Emacs with a purely text-based format, such as you would use with a virtual terminal (which is my preference), you can start Emacs using the **-nw** (no window system) option:

```
emacs -nw
emacs -nw document
emacs -nw document names addresses phone-numbers
```

■ **Note**

If you prefer the Emacs text-based format, here is a very cool trick I think you will like. What you are about to read is highly technical, so if you don't understand it, you can ignore it.

When you start Emacs using the following command, the program runs in an extra-large window:

```
printf '\e[8;50;100t' ; emacs -nw
```

The details of how this works will take us too far afield, so I won't explain it all. Suffice to say that the **printf** command above changes the size of the terminal window, and the **;** (semicolon) separates two commands that are executed one after the other.

Of course, you wouldn't type this command every time you want to use Emacs. Instead, you can create an alias and use that to start the program. The following command creates a suitable alias named **e**:

```
alias e="printf '\e[8;50;100 t' ; emacs -nw"
```

To make this convenient, all you have to do is put the **alias** command in your **.bashrc** file. (For a discussion of dotfiles, see Section 2.18.)

Once that is done, you can start a text version of Emacs in a large terminal window whenever you want simply by entering:

```
e
```

Section 5.3: Starting Emacs as a Read-Only Editor

There may be times when you want to use Emacs to look at an important file that should not be changed. To do this, start Emacs using the following syntax:

emacs *file* **-f read-only-mode**

For example, say that you need to look at a file named **secrets**. You want to use Emacs to look at the file, but you want to be sure you don't accidentally make any changes to it. Enter the command:

```
emacs secrets -f read-only-mode
```

When Emacs starts, the file will be marked as being read-only. This means you can look at it, but not make any changes.

■ **Note** When you start Emacs by using the **-f** option, it tells Emacs to execute the Lisp function that is specified after the option. In this case, you are telling Emacs to execute a function named **toggle-read-only**. This function tells Emacs to consider the file you are editing as being read-only.

Section 5.4: Recovering Data After a System Failure

As a general rule, if your work is interrupted abnormally, you stand to lose all the data you entered since the last time you saved. For this reason, it is a good idea to pause and save your work regularly.

Still, accidents do happen, and Emacs works behind the scenes to protect you from losing data in case of a system failure. Whenever you are editing a file, Emacs automatically saves a copy of that file at regular intervals (by default, every 300 keystrokes). This backup file is called the AUTO-SAVE FILE.

Emacs creates the auto-save file in the same directory as the file you are editing. (We discussed directories in Section 2.15.) The name of the file will be the same as the file you are editing, except that there will be a **#** (number sign) character at the beginning and end of the name. For example, if you are editing a file named **document**, Emacs will create an auto-save file named **#document#**. So if you are looking at a directory and you see a file with such a name, you will understand what it is and how it got there.

Whenever you save a file, Emacs automatically removes the auto-save file. If you make more changes to the file, Emacs creates a new auto-save file. Thus, under normal circumstances, you should never see an auto-save file. However, if your Emacs session is terminated abnormally — before you have a chance to save your work — the auto-save file is preserved.

Each time you tell Emacs to begin work with an existing file, Emacs first checks to see if there exists a corresponding auto-save file. If so, it means that the last time you edited the file, you were unable to save your work properly. In such cases, Emacs will display a message like the following:

```
Auto save file is newer; consider M-x recover-file
```

This means Emacs thinks that you need to recover your file. To do so, simply follow the instructions. Enter the command:

```
M-x recover-file
```

Emacs will display the name of the original file. If this is correct, press <Enter>. Emacs will now ask your permission to restore the auto-save file. At the same time, Emacs will create a new window and display the directory information for the original file. (We discussed directories in Section 2.15.)

If you decide to restore the file, type **yes**. Emacs will replace the text you are editing with the contents of the auto-save file. If you do not want to restore the auto-save file, just type **no**.

To summarize, if your work is interrupted abnormally, Emacs will probably have saved what you were doing in an auto-save file. If so, you will see a message suggesting that you recover the file the next time you start to edit that file. To recover the file, enter:

```
M-x recover-file
```

When Emacs displays the name of the file, press <Enter>. Then, when Emacs asks for permission to restore the file, enter **yes**.

Section 5.5: Stopping Emacs

To stop Emacs, use the command **C-x C-c**. If there is no need to save any files, you may see a message like this:

```
(No files need saving)
```

Regardless, Emacs will quit, and you will be returned to the shell prompt.

If you have been working with one or more files that have not yet been saved, Emacs will give you a chance to save your work before it quits. For each file that has not been saved, Emacs will display the name of the file along with several choices.

Here is an example. You have been working with a file named **document** that has not as yet been saved. You press **C-x C-c** and you see the following:

```
Save file /home/harley/document? (y, n, !, ., q, C-r or C-h)
```

Emacs is asking if you want to save the file. Notice that Emacs displays the full pathname of the file:

```
/home/harley/document
```

This shows the file name and the directory in which it resides (see Section 2.17).

At this point, you have a number of choices. Most likely, you will want to save the file and quit. To do so, press **y** (for yes). If you don't want to save the file, press **n** (for no), and Emacs will quit without saving. Be careful: if you press **n**, any changes you have made to the file since the last time you saved it will be lost.

■ **Note** If you are editing an existing file and you happen to make a lot of mistakes, you may decide to abandon everything you have done. Simply press **C-x C-c**, and then press **n** to quit without saving.

When you are editing more than one file, Emacs will display the name of each file that needs to be saved, one at a time, and ask you what to do. As before, you can press **y** to save and **n** to not save. However, you also have a few other choices.

To save all of the files at once and then quit, press **!** (exclamation mark). To quit immediately, without saving anything more, press **q**. To save the current file only, but quit without saving anything else, press **.** (period).

If you try to quit when there are still files that are not saved, Emacs will ask you to confirm your intentions. You will see a message like:

```
Modified buffers exist; exit anyway? (yes or no)
```

In this case, you must enter either the full words **yes** or **no**.

Notice that the message refers to "buffers". I will explain what they are in a moment. For now, you can consider each buffer to be a separate working area.

Note When Emacs wants to be especially sure you are making the right decision, it will ask you to answer **yes** or **no** (as opposed to **y** or **n**). In such cases, you must type the full word. This prevents you from making a serious mistake by accidentally pressing a single wrong key.

When you are working with only a single file, you can see the file on your screen when Emacs asks if you want to save it. However, when you are editing more than one file, Emacs will ask you about each file in turn, and you may have forgotten what was in one of the files. If so, when Emacs asks what to do with that particular file, press **C-r**. Emacs will show you the file, so you can make an informed decision. As you are looking at the file, you will be in VIEW MODE, which means you can read the file, but not make any changes. To quit viewing the file, press **q** (for quit).

Finally, if you forget what any of the choices mean, press **C-h** to display a quick help summary. To get rid of the help information, press **q**.

To summarize, you stop Emacs by pressing **C-x C-c**. If there are files to be saved, you have the choices summarized in Figure 5-1.

y	Save the specified file
n	Do not save the specified file
!	Save all the remaining files
q	Quit immediately without saving
.	Save the specified file and then quit
C-r	View the specified file
C-h	Display help information

FIGURE 5-1. Choosing whether or not to save files.

When you tell Emacs to quit, and you have files that have not been saved, Emacs will ask you what to do with each file in turn. Here are the responses you can use to make a choice for each such file.

CHAPTER 6

Commands, Buffers, Windows

Section 6.1: Commands and Key Bindings

In Section 4.2, I introduced the term "key sequence". However, at the time, I didn't give you a strict definition. Nor have I, as yet, given you a strict definition of an Emacs "command". However, as the poet observes:

"The time has come," the Walrus said,
"To talk of many things:
Of bindings— using Lisp commands—
Of Emacs' useful rings—
Of sequences that make you frown—
And whether keys have wings."

So, are you ready? Take a deep breath.

You know by now that the basic way you use Emacs is that you press one or more keys and something happens. Here is why.

Every Emacs COMMAND is a Lisp function (program module) that has been designed to be used interactively. All commands have a name, usually consisting of lowercase letters and hyphens. For example, the command that moves the cursor up one line is called **`previous-line`**. The command that moves the cursor down one line is called **`next-line`**.

An INPUT EVENT is anything you can do to send input to the computer: pressing a key, clicking a mouse button, moving a mouse, and so on. A KEY SEQUENCE is a series of input events that, taken as a unit, have meaning. Key sequences are used to invoke commands.

The connection between a key sequence and the command it invokes is called a KEY BINDING. Once such a connection exists, we say that the key sequence is BOUND to that specific command. For example, the key sequence **`C-p`** is bound to the **`previous-line`** command; **`C-n`** is bound to the **`next-line`** command; **`C-f`** is bound to the **`forward-char`** command; **`M-f`** is bound to the **`forward-word`** command; and so on.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_6

Whenever you type a key sequence, you are actually telling Emacs to execute the command that is bound to that sequence. So, to use a specific command, you type the key sequence that is bound to that command. In our example, when you want to move the cursor up one line, you press **C-p**. This tells Emacs to execute the **previous-line** command. When you press **C-n**, you tell Emacs to execute the **next-line** command.

Here is another example. To move the cursor forward one character (that is, one position to the right), you press **C-f**. This invokes a command called **forward-char**. To move the cursor forward one word, you press **M-f**. This invokes a command called **forward-word**.

There are hundreds of Emacs commands and — in one sense — learning to use Emacs means learning how to use all the key sequences that are bound to the most useful commands. Specifically, to learn how to edit text with Emacs, you will need to memorize the key sequences for moving the cursor, displaying text, making changes, deleting characters, manipulating buffers, using windows, loading and saving files, and so on.

Most of the time, you don't need to know the names of the actual commands you are using: all you need to remember is which keys to press. However, there is a reason I am telling you all this. Emacs has so many commands that there aren't enough key sequences to go around, which means that many Emacs commands do not have their own key sequence. To use such commands, you have to type the full name of the command. To do so, you press **M-x**, type the name of the command, and then press <Enter>. Emacs will then execute that command.

Here is an example. The **ispell-buffer** command helps you check the spelling of all the words in your buffer. (We will discuss buffers in Section 6.2. Basically, a buffer is a work area.) The **ispell-buffer** command is not bound to any particular key sequence. Thus, you cannot run it by typing a key sequence. Instead, you must press **M-x**, then type **ispell-buffer** and press <Enter>.

Here are several experiments you can try for yourself. First, you might be wondering, can I use **M-x** to execute any command by specifying its full name rather than pressing its key? Of course.

For example, I mentioned that **C-p** moves the cursor to the previous line by executing the **previous-line** command. Try this. Start Emacs and type a few lines of text. Now press **C-p**. Notice that the cursor moves up one line. Now press **M-x**, type **previous-line,** and then press <Enter>. Again, the cursor moves up one line. Of course, you would never move the cursor by typing **M-x previous-line**, because pressing **C-p** is so easy. However, this example does show how it all works.

Here is another, more interesting example. Emacs comes with a number of games and diversions you can use by executing the appropriate programs. One such game is **doctor**: a program that acts like a therapist. (This is actually a modern version of a program called Eliza that was written many years ago at MIT.)

I will discuss the Emacs games in Section 12.7. For now, though, you may want to try talking to the built-in Emacs therapist. Press **M-x**, type **doctor**, and then press <Enter>. (Note: If Emacs displays a message saying **[No match]**, it means this program is not installed on your system.)

Once **doctor** starts, the two of you take turns talking. You can type as much as you want. When you are finished talking, press <Enter> twice, and the program will respond. When you are ready to quit, press **C-x k**, and then press <Enter>. (The **C-x k** command kills the buffer in which the program is running.)

One last point. You might wonder, why does Emacs use such long command names? After all, long names are hard to memorize and slow to type, and Emacs was designed for users who get frustrated with tools that won't let them think quickly. The answer is, when you are called upon to type a command name, Emacs will help you, so you rarely have to type the entire name. This facility is called "completion", and I will show you how it works in Section 6.7.

Note

Understanding the idea of key binding is important for several reasons. First, key binding is a basic part of Emacs, and until you understand it, you won't really understand Emacs.

Second, if there is a command you find useful but the key sequence is awkward for you to use, you can change it to a key sequence you like better.

Finally, as I mentioned above, there are many commands that do not have their own key sequences. To use these commands, you need to invoke them manually, using **M-x**. Once you understand key binding, you can learn how to bind a specific key sequence to any command you want, which makes it much easier to use that command.

To be sure, being able to customize your work environment in this way is an advanced skill, but it all starts with a firm understanding of key binding.

Section 6.2: Buffers

One of the nice features of Emacs is that it lets you do more than one thing at a time. For example, you can edit as many files as you want, jumping from one to another as the mood takes you.

In order to offer this flexibility, Emacs keeps a separate storage area, called a BUFFER, for each particular task. For example, if you are editing three different files, Emacs will maintain three separate buffers, one for each file.

Understanding buffers and how to use them is a crucial skill you must develop in order to become comfortable with Emacs. So let's take a few moments to explore what these things are and just how they work.

There are two ways in which buffers are created. First, you can make a new buffer whenever you want. Second, Emacs will automatically create a buffer when the need arises.

The thing to remember is that everything you see and everything you type is kept in one buffer or another. For example, Emacs contains a comprehensive, built-in help system called the Info facility that you can use whenever you want. When you press the key sequence to ask for help (it happens to be **C-h i**), Emacs creates a new buffer to hold the help information.

Here is another example. In Section 6.1, I mentioned that Emacs comes with a program called **doctor** that acts like a therapist. (You tell it your problems, and it responds with meaningless platitudes.) When you enter the command to start it (**M-x doctor**), Emacs creates a new buffer in which to run the program.

One last example. To keep track of your resources, you can use a command that tells Emacs to display a list of all your buffers (the command is **C-x C-b**). When you use this command, Emacs creates yet another buffer to hold the actual list as it is being displayed.

To keep track of all your buffers, Emacs assigns each one of them a unique name. When Emacs is called upon to create a buffer on its own, it will choose an appropriate name. For example, when you start the Info facility, Emacs creates a buffer named ***info***, and when you run the **doctor** program, Emacs creates a buffer named ***doctor***. When you start editing a file, Emacs creates a buffer with the same name as the file. Thus, if you tell Emacs you want to edit a file named **document**, it will create a buffer named **document** to hold that file.

At all times, Emacs makes sure that you have at least one buffer. When you start Emacs by specifying a file to edit, Emacs will create a buffer by that name. If the file exists, Emacs will read its contents into the buffer. If not, Emacs will create a brand new file by that name, and the buffer will be empty. For example, to start working with a file named **document**, you enter the command:

```
emacs document
```

When you start Emacs without a file name:

```
emacs
```

you will find yourself with an empty buffer named ***scratch***. Since Emacs does not know what file you want to use, it creates the ***scratch*** buffer, so you will have someplace to work.

One of the most important uses for a buffer is to act as a temporary work area when you need to make some quick notes. For example, say that your mother calls you on the phone as you are working on an essay. She tells you to write down the name of a wonderful book you should read (*Harley Hahn's Guide to Unix and Linux)*, but you don't want to take the time to look for a piece of paper and a pen. Instead, you quickly create a new buffer and type the information. Once your mother hangs up, you switch back to the buffer that contains the essay you are typing. The new buffer remains hidden from view, where you can deal with it at your leisure.

In Section 6.2, I will discuss the commands you can use to manipulate your buffers. For now, just remember the following five important ideas:

- Everything you do with Emacs is contained in a buffer.
- Each buffer has a unique name.
- You can create a new buffer whenever you want.
- You can kill (delete) a buffer whenever you want.
- Some buffers are created by you; some are created automatically by Emacs.

Note Buffers are your friends.

Section 6.3: Windows

As we discussed in Section 6.2, everything you do with Emacs takes place within a buffer. You can have as many buffers as you want and, much of the time, you will have several things going on at once.

But how do you see what is in your buffers? The answer is that Emacs creates one or more WINDOWS on your screen and, within each window, you can view the contents of a single buffer. Some people prefer to use one large window and look at only one buffer at a time. Other people like to use multiple windows to look at more than one buffer at a time. For example, say that you are working with three different files. You might decide to have three windows, each of which displays a different file in its own buffer.

As you become experienced with Emacs, you will develop your own personal style. Most of the time, you will probably use just one or two windows, creating and deleting extra ones as the need arises.

So you can see what it looks like, Figure 6-1 shows a typical Emacs screen with a single window; Figure 6-2 shows a screen with two windows.

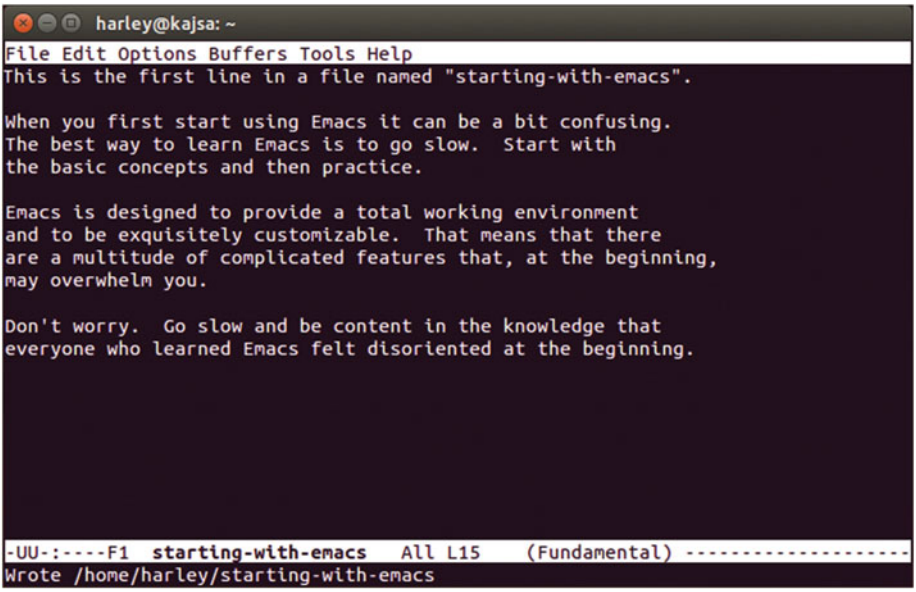

FIGURE 6-1. A typical Emacs screen with a single window.

This is the simplest Emacs configuration: one window displaying the contents of a buffer.

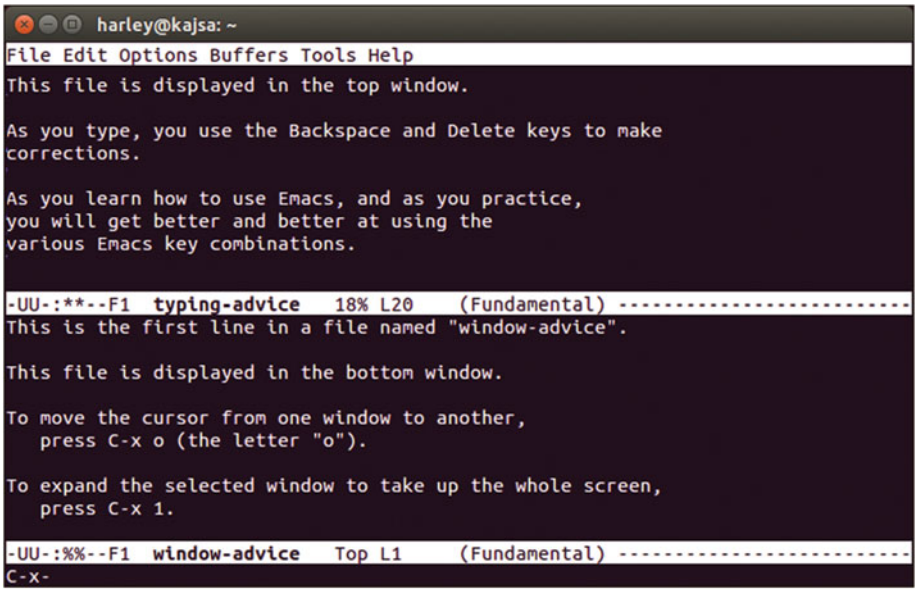

FIGURE 6-2. A typical Emacs screen with two windows.

Emacs lets you use multiple windows, each of which displays the contents of one buffer. In this example, there are two windows displaying two different buffers.

The best way to think of a window is as a fixed-size opening into a buffer. When you look into a window, you are looking at the part of the buffer that is currently being displayed. If you want to look at another part of the buffer, you can move the window up or down (or even sideways).

One nice thing about Emacs is that it lets you display a particular buffer in more than one window at the same time. This comes in handy when the content of the buffer is too large to fit into a single window. In such cases, you can use two windows to look at different parts of the same buffer at the same time.

For example, say that you are editing a long document. You can display the beginning of the document in one window and the end of the document in another window. This makes it easy to copy or move text from one part of the buffer to another, without having to jump around and lose your place.

Here is something interesting. Let's say that you have two windows, each of which is displaying the same part of a particular buffer. What do you think will happen if you make a change to the text in one of the windows? Well, since each window is showing you the same buffer, changing the text in one window should affect the text in the other window and, indeed, that is what happens. As you type or edit the text in one window, you can see both windows change at the same time.

To see how this all works, try it for yourself. Start Emacs by using the **`-Q`** option to suppress the splash screen (see Section 5.1), so it does not get in the way of the **`*scratch*`** buffer. Either of the following commands (see Section 5.2) will do:

```
emacs -Q
emacs -Q -nw
```

Emacs will start by creating one large window containing a buffer named **`*scratch*`**. There will be a few lines at the top with some useful information. To erase these lines, use **`C-x h C-w`**. (**`C-x h`** selects the entire buffer; **`C-w`** erases the selection.)

Now create a duplicate of the window by pressing **`C-x 2`** (the command is explained in Section 7.6. You should now have two windows, each of which shows the empty **`*scratch*`** file.

Start typing anything. Notice that everything you type shows up in both windows. Press the Backspace key a few times to erase the most recently typed characters. Notice that, as you erase a character, the change is updated in both windows. Take a moment to experiment by making some more changes, and see what happens.

When you are finished, press **`C-x C-c`** to stop Emacs.

As you work with Emacs, the cursor is always in one particular window, which we call the SELECTED WINDOW or CURRENT WINDOW. As you type, the characters are inserted into the selected window at the position of the cursor. If you want to insert characters into a different window, you must first move the focus to that window. (I will explain how to do this in Section 6.3) Once you do, the cursor will move to that window and it will become the selected window.

Section 6.4: The Mode Line / Read-Only Viewing

Near the bottom of each window is a special line called the MODE LINE. The mode line contains information about the buffer that is currently being displayed in that window. The mode line is displayed in reverse colors to make it easy to read.

Take another look at the screenshot in Figure 6-1. This screen has a single window and, hence, one mode line near the bottom of that window. You can see this mode line, by itself, in Figure 6-3. In Figure 6-2, there are two windows, so there are two mode lines, which you can see in Figure 6-4.

FIGURE 6-3. The mode line from Figure 6-1.

In Figure 6-1, there is only one window. Thus, there is only one mode line.

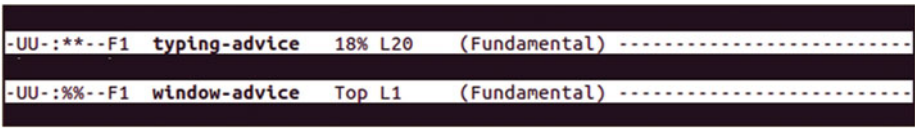

FIGURE 6-4. The two mode lines from Figure 6-2.

In Figure 6-2, there are two windows, so there are two mode lines, one for each window.

Notice that near the left-hand side of each mode line there is a colon. The two characters after the colon show you the status of the buffer. The meaning of these characters is shown in Figure 6-5. In our first example, the single mode line, the two characters are `--` (two hyphens). This indicates either that the buffer has not been modified at all or has not been modified since the last time you saved the file. When you see two hyphens in this position on a mode line, it means that, if you were to quit now, there would be nothing to save.

Characters	Meaning
`--`	Buffer has not been modified
`**`	Buffer has been modified (not yet saved)
`%%`	Read-only mode: buffer has not been modified
`%*`	Read-only mode: buffer has been modified

FIGURE 6-5. Status characters within the mode line.

On the mode line, to the right of the colon, Emacs displays two characters to show you the status of the buffer.

In the next example, the first mode line in Figure 6-4, the two characters are **** (two asterisks). This means that the buffer has been modified in some way. Whenever you see ****, it reminds you that you must save the contents of the buffer before you quit.

In the final example, the second mode line in Figure 6-4, the two characters are `%%` (two percent signs). This means that the buffer is in read-only mode and has not been modified.

At any time, you can turn read-only mode on and off by typing **C-x C-q** to execute the Emacs command **read-only-mode**. For example, if you are editing a file in read-only mode, and you decide you want to be able to make changes, type **C-x C-q**. To change back to read-only mode, type **C-x C-q** again.

Note

As we discussed in Section 5.3, when you start Emacs with the name of a file, you can tell Emacs you want to work in read-only mode by using the **-f** option to execute the Lisp function **toggle-read-only**. For example, to edit a file named **secrets** in read-only mode, you can use the command:

```
emacs -f toggle-read-only secrets
```

If you start with a file in read-only mode in this way and you decide you want to be able to make changes, just type **C-x C-q**.

To continue: the next item of information on the status line is the name of the buffer. In our examples, you can see the names are **starting-with-emacs**, **typing-advice**, and **window-advice**. These are the names I chose when I created the buffers. As we discussed in Section 6.2, Emacs sometimes creates buffers on its own. In particular, when you start Emacs without specifying a file name, Emacs will create an empty buffer named ***scratch***. Also, when you start the Info facility, Emacs creates a buffer named ***info*** to hold the help information.

To the right of the buffer name is information that gives you a rough idea of the current position within the buffer. If the entire buffer is small enough to be contained within the window, you will see **All**. This is the case in our first example.

If the buffer is too large to fit into the window all at once, you will see three possible position descriptions. If the window is currently showing the beginning of the buffer, you will see **Top**. (This is the case in our third example.) If the window is showing the end of the buffer, you will see **Bot** (bottom). Otherwise, you will see a number. This number indicates what percentage of the buffer is above the top in the window. In our second example, 18 percent of the buffer is above what we see in the window.

To the right of this, you will see the letter **L** followed by a number, indicating the line number of the cursor position within that buffer. In our first example (Figure 6-3), the cursor is on line 15.

Finally, to the right of the line number, you will see one or more words in parentheses. These words show you the mode in which Emacs is operating for that particular buffer. I will talk about modes in Section 11.1. For now, I will just say that Emacs can act in different ways to suit the type of work you are doing. For example, if you are editing English text, Emacs will work a little differently than if you are writing a Lisp program. The mode shows what personality Emacs is using at the moment. (This, by the way, is where we get the name "mode line".) In all three of our examples, Emacs is in Fundamental mode (which I will explain in Section 11.2).

Section 6.5: The Echo Area / Typing Emacs Commands

When you type something and you instantly see it on your screen, we say that the program you are using ECHOES what you are typing. For example, when you type a Unix command at the shell prompt, the shell echoes the command. This means that the shell displays each character as you type it.

Emacs is different: it only echoes some commands. When it does, the characters are displayed on the bottom line of your screen, which is called the ECHO AREA. Here is how it works.

Emacs does not echo any commands that consist of only a single character combination. For example, the command **`C-n`** moves the cursor to the next line. When you press this key sequence, you do not see the letters **`C-n`** in the echo area: all you see is that the cursor moves.

Emacs echoes only multi-character commands. However, it waits one second after you press a character before it echoes it. If, within that time, you press a second character, Emacs does not echo the first one.

For example, the command **`C-x k`** kills (deletes) the current buffer. When you type the first character (**`C-x`**), Emacs waits for you to type another one. If you do not type another character within one second, Emacs echoes what you have already typed. That is, the letters **`C-x`** will appear on the bottom line of the screen. However, if you complete the command quickly, nothing will echo. Emacs will simply carry out the command.

The reason that Emacs echoes in this way is to give quick typists a fast response without a distraction, while providing slower, more hesitant typists with as much feedback as possible. In practice, this means that when you type those commands with which you are the most familiar, things move fast, but when you type the commands that are still new to you, Emacs prompts you as necessary. The overall feeling is that the system speeds up or slows down to match your comfort level. (Think about that for a moment.)

The echo area is also used by Emacs to display messages. These may be error messages, warnings, or simply informative comments. Whenever it displays an error message, Emacs will make a sound to make sure to get your attention.

If you look at Figures 6-1 and 6-2, you will see examples of how Emacs uses the echo area. At the bottom of the screen in Figure 6-1 is the following message:

```
Wrote /home/harley/starting-with-emacs
```

This message tells us that Emacs has successfully saved the contents of the buffer to a file. At the bottom of the screen in Figure 6-2, you can see:

```
C-x-
```

This means that you have pressed the **C-x** key and that Emacs is waiting for you to type something else to complete the command.

As you might expect, you can type ahead as much as you want. That is, you can press the keys as fast as you can, and Emacs will remember what you type. However, when you make a mistake that generates an error message, Emacs will throw away all the pending keystrokes. This prevents a mistake from causing unexpected problems.

Finally, if you are typing a command and you change your mind, press **C-g** to cancel the command. We will discuss this in Section 6.6.

Section 6.6: The Minibuffer

Many commands require you to enter further input once you press the initial key sequences. For example, the command **C-x C-f** tells Emacs to read a file into a buffer. Once you press **C-x C-f**, Emacs will ask you for the name of the file. If the file exists, Emacs will copy it into a new buffer. If the file doesn't exist, Emacs will create it for you.

When Emacs displays a message asking for information, it writes it to the bottom line of your screen. You are expected to type the information and then press <Enter>. Whatever you type is echoed on the bottom line of the screen and, up until the time you press <Enter>, you can make corrections.

Thus, the bottom line of your screen has two purposes: First, as I explained in Section 6.5, Emacs uses this line to echo your regular keystrokes and to display messages. Second, Emacs uses this same line to ask you for information and to echo such information as you type it.

For this reason, the bottom line of your screen has two different names. When Emacs is echoing your commands or displaying messages, this line is called the echo area. When Emacs is asking you for information and reading your reply, this line is called the MINIBUFFER.

As you type information into the minibuffer, you can use the Backspace key to correct mistakes. Each time you press <Backspace>, it erases one character.

As an Emacs user, you will often find yourself typing information into the minibuffer. To help you, Emacs does two things that make your life easier. First, whenever possible, Emacs will display a default value in parentheses when it prompts you for information. This default value is Emacs' guess as to what you might want to type. If indeed this is what you want, all you need to do is press <Enter>. Otherwise, you can type a different value.

Here is an example. You are working with three different buffers: **names**, **addresses** and **phone-numbers**. At the current time, you are editing the **names** file, and you decide to switch to **addresses**. The command to change to another buffer is **C-x b**. As soon as you type this, Emacs displays the following in the minibuffer (the bottom line of your screen):

```
Switch to buffer (default addresses):
```

Emacs is asking you for the name of the buffer to which you want to switch. The message in parentheses is telling you that the default value is **addresses**.

Thus, if you want to switch to the **addresses** buffer, all you need to do is press <Enter>. If you want to switch to another buffer (such as **phone-numbers**), you can type its name and then press <Enter>.

The second way in which Emacs makes it easy to enter information into the minibuffer is a facility called COMPLETION. Completion is a process by which you can tell Emacs to guess what you are going to type, so you don't have to actually type all the characters yourself. Completion is an important topic, and we will discuss it in more detail in Section 6.7.

■ **Note** Emacs uses the bottom line of your display for both the minibuffer and echo area. The difference is that the minibuffer is used for input, while the echo area is only for output.

Occasionally, Emacs will need to display something (such as an error message) while you are typing in the minibuffer. When this happens, Emacs will display the message and the minibuffer will disappear temporarily. After a few seconds, Emacs will erase the message and the minibuffer will reappear. In other words, the bottom line of your screen will have been transmogrified from the minibuffer into the echo area and then, after a few seconds, back to the minibuffer.

Occasionally, you will be typing in the minibuffer when you realize that you are making a big mistake. Emacs makes it easy to cancel the whole command: all you need to do is press **C-g** (before you press <Enter>).

This is a key worth remembering: **C-g** within Emacs acts a lot like the **^C** (intercept) key within Unix (see Section 2.11). One day this may save your life.

■ **Note** If you get yourself into a situation within Emacs where things are getting weird and you don't know what to do, try pressing **C-g** to cancel the current command. If it doesn't work, press **C-g** a second time.

Section 6.7: Completion

One of the nice ways Emacs makes your minute-to-minute work easier is that, at various times, you can tell Emacs to guess what you are going to type and complete it for you. This facility is called COMPLETION and here is how it works.

Whenever you are typing in the minibuffer — that is, providing information in response to a prompt — you can use one of the completion commands (explained below). This is a signal to Emacs to try to complete what you are typing. Emacs will display what it thinks you want to type. If Emacs has guessed correctly, all you have to do is press <Enter>. Otherwise, you can make whatever correction is necessary and then press <Enter>.

For example, let's say that you would like to switch to a different buffer. At the current time, you have three buffers called **names**, **addresses** and **phone-numbers**. Right now, you happen to be editing the **names** buffer, but you want to switch to the **phone-numbers** buffer.

The command to switch to a different buffer is **C-x b**. When you type this command, Emacs will display a prompt in the minibuffer. In this case, you might see:

```
Switch to buffer (default addresses):
```

This means that Emacs is asking you for the name of the buffer to which you want to switch. The default is **addresses**, so if this were the buffer you want, you would press <Enter>.

In this case, however, you want to switch to a different buffer, **phone-numbers**. You could type the entire name and then press <Enter>. The shortcut, though, would be to type only a **p** and then type a completion command. Emacs will then guess what you want, and type the rest of the name for you. In this case, you would see:

```
Switch to buffer (default addresses): phone-number
```

Now all you have to do is press <Enter>.

The completion facility — like much of Emacs — has a lot of complex details. However, all you really need to know are the four completion commands. They are all single keys: **TAB** (the Tab key), **SPC** (the Space key), **RET** (the Enter key), and **?** (question mark). Figure 6-6 summarizes how these keys work.

Command	Action
TAB	Complete text in minibuffer as much as possible
C-i	Same as **TAB**
SPC	Complete text in minibuffer up to end of word
RET	Same as **TAB**, then enter the command
?	Create new window, display list of possible completions

FIGURE 6-6. Completion Commands.

When you type in the minibuffer, Emacs helps you, whenever possible, by guessing what you want and letting you use completion commands to make choices. **TAB** *and* **?** *work for commands, files and buffer.* **SPC** *works for commands and buffers.* **RET** *works only for commands.*

The **TAB** command works for buffers, commands and files. Most of the time, **TAB** is all you will need. **TAB** tells Emacs to complete as much as possible and then wait for you to press <Enter>. Here is how it would work in our example. Emacs has just displayed the message:

```
Switch to buffer (default addresses):
```

You want to switch to the **phone-numbers** buffer, so you type the single letter **p** and press <Tab>. Emacs looks through its list of buffers to see if any of them begin with **p**. In this case, it happens there is only one, the **phone-numbers** buffer. So Emacs types the rest of the name for you. You can now press <Enter> and complete the command.

The **SPC** command works for buffers and commands, because file names can contain spaces (although it is a bad idea; see Section 2.18). **SPC** is similar to **TAB** except that **SPC** will only complete up to the end of the word. In this case, if you had pressed <Space> instead of <Tab>, Emacs would have completed only up to the hyphen after the word **phone**, and you would see:

```
Switch to buffer (default addresses): phone-
```

To complete the command, you would have to either type the rest of the name, or type **n** and press <Tab> or <Space>.

SPC is handy when you want to use part of a name and finish the rest for yourself. For example, say that you wanted to create a new buffer named **phone-messages**. You can press **C-x b**, then press **p** and **SPC** to get the response described above. Now you can type **messages** and then press <Enter>. Since this buffer does not already exist, Emacs will create it for you. In this way, you create a brand new buffer named **phone-messages**.

RET works like **TAB** with the added effect that, after the completion is finished, Emacs will enter the command for you automatically. Thus, pressing <Enter> is like pressing **TAB** followed by <Enter>. In other words, if you are sure that there is only one way for Emacs to complete what you are typing, you can simply press <Enter> instead of **TAB** <Enter> and save yourself a keystroke. **RET** works only for commands, because when you type a brand new buffer name or file name, you need to be able to press <Enter> to indicate the end of the name.

The last completion key is **?** (a question mark) which, like **TAB**, works for buffers, commands and files. If you press **?**, Emacs will create a new window and, within it, display a list of all the possible completions it can find. This is handy when there are a number of completions, and you are not sure which one you want. Once you see the one you want, you can type it and press <Enter>. If you don't see the one you want, you can press **C-g** to cancel the command.

Here is an example. There is a command named **auto-fill-mode** which is useful when you are typing English text. It tells Emacs to break long lines for you automatically as you type, so you don't have to keep pressing <Enter>. To execute this command, you would press **M-x**, type **auto-fill-mode**, and then press <Enter>.

Let's say that you want to execute this command, but you forget the full name. You remember that it starts with the word **auto**, but that is all. Press **M-x** and then type **auto**. The minibuffer now looks like this:

```
M-x auto
```

Now press the **?** key. Emacs will create a new window with a buffer named ***Completions***. Within this window you will see all the possible completions. In this case, there are two:

```
auto-fill-mode
auto-save-mode
```

To finish the command, all you need to do is type **-f** and press <Enter>. Since there is only one possible completion, Emacs will type it for you and enter the command.

The power of the completion facility lies in the fact that you can use it just about any time you are called upon to type something into the minibuffer. You can use completion for file names, buffer names, Emacs commands, and so on. (Moreover, as I mentioned in Section 2.12, you can use some of the same completion shortcuts when you are entering Unix commands for the shell.) Once you get used to the completion keys — and remember, most of the time, all you really need is **TAB** — you will come to appreciate the power of Emacs.

Now you can see why it is okay that so many Emacs commands have long names. Most of the time, you never really have to type the full name.

■ **Note**

When you use the **?** completion command, Emacs creates a new buffer named ***Completions*** and, within it, displays a list of all the possible completions. If you want, you can make a choice by changing to the ***Completions*** window, moving to the selection you want, and pressing <Enter>. (I explain how to change from one window to another in Section 7.6.)

However, most of the time, it is easier to complete the command on your own by typing a few more keystrokes and then pressing either <Tab> or <Enter>.

Section 6.8: Disabled Commands

This is an advanced topic. I do want you to read this section, but if everything doesn't make sense to you right away, don't worry about it. When you need the information, it will be here waiting for you.

Some Emacs commands are troublesome for beginners because they can be confusing or even potentially destructive. To protect you, such commands are marked as being DISABLED. When you use a DISABLED COMMAND, Emacs will pause before it runs the command. You will see a message telling you the command is disabled, and you will be asked if you really want to run the command. You can then indicate yes or no. If you become comfortable with the command, you can ENABLE it permanently, which lets you use it whenever you want without looking at the warning. Here is an example to show you how it works.

In my experience the disabled commands you are most likely to encounter are two commands used to change the case of a specific area of text:

- **C-x C-u** converts text to all uppercase letters. This key sequence is mapped to the command **upcase-region**.
- **C-x C-l** converts text to all lowercase letters. This key sequence is mapped to the command **downcase-region**.

We will discuss these commands and how to use them in Section 8.8. For now, I'm just going to use them as examples.

When you type **C-x C-u,** Emacs displays the following message:

```
You have typed C-x C-u, invoking disabled command upcase-region.
It is disabled because new users often find it confusing.
```

Following this will be a technical description of the command. Read this carefully and decide if you really want to run it. Next you will see a question:

```
Do you want to use this command anyway?
```

This question is followed by a list of possible responses, shown in Figure 6-7. All you have to do is press one key to indicate your answer. If you want to run the command, press **y**. If you didn't really mean to use this command, press **n**. If you are confused, type **C-g**, the Emacs keyboard-quit command (see Section 7.4).

y	Run command; enable it for rest of the work session
n	Do not run command; leave it disabled
SPC	Run command once; leave it disabled
!	Run command; enable *all* commands for the rest of the work session

FIGURE 6-7. Choosing whether or not to run a disabled command.

*When you type a disabled command, Emacs tells you the command is disabled and asks you what you want to do. Most of the time, you will answer either **y** or **n**. Note: You don't need to press <Enter>.*

Note

One reason **C-x C-u** is disabled is that it is very similar to **C-x u**, the Emacs **undo** command. This is the command you use to reverse changes to the buffer (see Section 7.3), so it is used a lot.

If you mean to type **C-x u** but you hold down the Ctrl key a bit too long, you will accidentally type **C-x C-u**. In such cases, it is helpful to see a warning message, rather than unexpectedly having the entire region change to all uppercase letters.

If there is a disabled command you are comfortable using, you can enable it permanently by putting the following line in your **.emacs** initialization file. (Using the .emacs initialization file is an advanced topic, which we will discuss in Section 11.8.)

```
(put 'command 'disabled nil)
```

where *command* is the command you want to enable.

For example, to enable the **C-x C-u** (**upcase-region**) and **C-x C-l** (**downcase-region**) commands, you would put the following lines in your **.emacs** file:

```
(put 'upcase-region 'disabled nil)
(put 'downcase-region 'disabled nil)
```

If you want to disable a command permanently, put the following line in your **.emacs** initialization file:

```
(put 'command 'disabled t)
```

where *command* is the command you want to disable. You might do this if there are commands you don't want to use by accident. You can still use them, but the automatic warning will slow you down.

As an example, to disable **C-x C-u** (**upcase-region**) and **C-x C-l** (**downcase-region**), put the following lines in your **.emacs** file:

```
(put 'upcase-region 'disabled t)
(put 'downcase-region 'disabled t)
```

As a convenience, there are two special commands that will add such lines to your .emacs file for you. They are **enable-command** and **disable-command**. Just type **M-x** (explained in Section 6.1) followed by either **enable-command** or **disable-command**. Then type the command you want to enable or disable, and press <Enter>.

For example, to permanently enable **C-x C-u**, type:

M-x enable-command upcase-region <Enter>

To enable **C-x C-l**, type:

M-x enable-command downcase-region <Enter>

This tells **enable-command** to put the appropriate lines in your **.emacs** file.

To permanently disable these commands, type:

M-x disable-command upcase-region <Enter>
M-x disable-command downcase-region <Enter>

Again, **disable-command** will put the appropriate lines in your **.emacs** file.

Finally, I will remind you that, for exact instructions on how to use the **C-x C-**u and **C-x C-l** commands, see Section 8.8.

CHAPTER 7

The Text Editing Work Environment

Section 7.1: How to Practice Using Emacs

In order to practice with Emacs, you need to start the program and (optionally) have some text to edit. Let me show you an easy way to do that. You can do this whenever you want to practice.

Start Emacs by using one of the commands below. The **`-Q`** option suppresses the splash screen (see Section 5.1) and, if you are using a terminal window, the **`-nw`** option runs Emacs in a text-based format (see Section 5.2).

```
emacs -Q
emacs -Q -nw
```

When Emacs starts, you will see one window displaying a buffer named **`*scratch*`**. The buffer will contain several lines showing an informative message.

To begin your practice session, press **`C-h`** to start the Help facility. At the bottom of the screen, in the minibuffer (see Section 6.6), you will see a message:

```
C-h (Type ? for further options)-
```

Press **`b`**. This tells Emacs to display information about key bindings (a list of all the Emacs key sequences and what they do).

In order to display this list, Emacs will create a new buffer named **`*Help*`**. Emacs will then split your screen into two windows. The top window will contain the **`*scratch*`** buffer; the bottom window will contain the **`*Help*`** buffer. In the bottom window, you will see the beginning of the key bindings list.

At this point, your cursor will be in the top window. (In other words, the top window will be the selected window, also called the current window. See Section 6.3.) What you want to do is get rid of the top window and work exclusively with the bottom window. To do this, press **`C-x 0`** (the number zero). This tells Emacs to kill (get rid of) the selected window.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_7

You will now have one large window containing the ***Help*** buffer, which consists of a long list of key bindings. This is a good place to practice using Emacs, because, while you are practicing, you will be examining a list of Emacs key sequences and the commands to which they are bound. There are lots of ways to move through a buffer. For now, all you need to know is to type **C-v** to scroll down one screenful and **M-v** to scroll up one screenful.

■ **Note** If you are using a terminal window and typing **M-v** (<Alt-V>) pulls down a menu, it is because your terminal window program is interpreting <Alt> as a menu key. For instruction on how to fix this problem, see Section 4.6.

If you are a beginner, simply skim through the list of key bindings, reading them and trying out whatever catches your interest. Don't worry about learning everything right away. I'll take you through all the important commands in due course. When you are finished with this list, you can type **C-x C-c** to stop Emacs.

If anything weird happens while you are exploring, you can type **C-h b** to get back the list of key bindings. If this doesn't solve the problem, just bail out by pressing **C-x C-c** and start all over again.

Once you learn how to insert and modify text, you will want to practice the editing commands. However, the ***Help*** buffer is read-only, so you can't make changes. To practice editing, start Emacs with the **-Q** option and use the ***scratch*** buffer as a work area. To summarize:

Here is how to read the list of key bindings:

1. Start Emacs using **emacs -Q** or **emacs -Q -nw**.
2. Press **C-h b** to display the information about key bindings.
3. Press **C-x 0** to remove the ***scratch*** window.
4. Look at the commands, and practice as much as you want.
5. When you are finished, press **C-x C-c** to quit.

And here is how to practice editing commands:

1. Start Emacs using **emacs -Q** or **emacs -Q -nw**.
2. Use the ***scratch*** window to practice as much as you want.
3. When you are finished, press **C-x C-c** to quit.

Section 7.2: Typing and Correcting

To create text, all you have to do is move the cursor to where you want the characters to be inserted and start typing. That's all there is to it. (In Section 8.2, I will show you how to move the cursor wherever you want.)

If you want to practice, start Emacs using one of the following commands:

```
emacs -Q
emacs -Q -nw
```

You will see a single window with a buffer called **`*scratch*`**. Just start typing. When you are finished, you can stop by pressing **`C-x C-c`**.

As you type, there are two ways to make simple corrections. To erase the character you have just typed (to the left of the cursor), press <Backspace>. To erase the character at the current position of the cursor, press <Delete> or **`C-d`**. In other words, <Backspace> erases to the left; <Delete> and **`C-d`** erase to the right. (These keys are described in Figure 7-1.) These key sequences are worth practicing. As the Roman philosopher Marcus Tullius Cicero said when he first learned Emacs: "*Cuiusvis hominis est errare, nisi mentis inops, in errore perseverare.*"[1]

Key	Action
`BS`	Delete one character to the left of cursor
`DEL`	Delete one character at the position of cursor
`C-d`	Same as **`DEL`**
`C-o`	Open a new line
`C-x u`	Undo the last change to the buffer
`C-_`	Same as **`C-x u`**
`C--`	Same as **`C-x u`**
`C-/`	Same as **`C-x u`**
`C-q`	Insert the next character literally

FIGURE 7-1. Keys to use while typing.

Note **`C-_`** is "<Ctrl>-underscore".

There are two ways to insert a new line. First, you can move to where you want the line, type whatever you want, and press the Enter key. Alternatively, you can move to where you want the line and press **`C-o`**. This will create a new line for you. (You can think of **`o`** as meaning "open".)

---

[1] Any man can make mistakes, but only an idiot persists in his error.

Here are the details for using **C-o**. If you are at the beginning of a line, **C-o** creates a new, empty line above the current line. If you are at the end of a line, **C-o** creates a new, empty line below the current line. If you are within a line, **C-o** breaks it into two separate lines. Experiment a little and it will all make sense.

As you read through this book, you will find many commands you can use to make changes in the buffer: to erase text, to replace text, to move text, and so on. As you use these commands, you will sometimes find yourself in the situation of having made a change that you really don't want. For example, you may have deleted a large chunk of the buffer and then realized that you made a horrible mistake.

In such cases, you can use the **C-x u** command to reverse the last change you made to the buffer. As you can see from Figure 7-1, there are three other key sequences you can use to do the same thing. Because this command is complicated, but important, I'll explain it in detail in Section 7.3.

To finish this section, I'll deal with an odd problem that comes up every now and then: entering a control character or other special character. For example, let's say you want to put an actual <Ctrl-C> or <Ctrl-H> into your text, the type of thing programmers need to do every now and then. The problem is that many of the Ctrl keys have special meanings in Emacs, as do Tab, Escape, Delete, and a few other keys.

To insert one of these characters into your text, first press **C-q**. This tells Emacs that the next character is to be taken literally. (Think of the **q** as meaning "quote"; that is, to take the next key you type literally.) Thus, to insert a <Ctrl-C> character, type **C-q C-c**. To insert a <Ctrl-Q>, type **C-q C-q**. To insert a tab, use **C-q TAB**.

When you type a Ctrl key into a buffer in this way, it will be displayed on your screen using the standard Unix notation: a **^** (circumflex) character representing <Ctrl>, followed by an uppercase letter (see Section 4.2). For example, if you type **C-q C-c**, you will see **^C** on your screen. If you type **C-q C-q** you will see **^Q**. Please remember that, even though you see two characters on your screen, they represent only a single character. (It is a good idea to become familiar with the Unix Ctrl-key notation. It is so widely used, you will see it even when you run Emacs under Microsoft Windows.)

Aside from these special keys, there is really no trick to entering characters when using Emacs: just type and be happy.

Section 7.3: The repeat and undo Commands; Redo

As you edit text, there are two needs that you will encounter frequently:

- Repeat the last command.
- Reverse the last change to the buffer.

To repeat the last command you entered, use the key sequence **C-x z** which is bound to the **repeat** command. To reverse that last change you made to the buffer, use **C-x u**, **C-/** or **C-_**, all of which are bound to the **undo** command. If you are used to

Microsoft Windows, you can think of **C-x z** as being like **^Y** and **C-x u** as being like **^Z**. However, as you will see, there are some differences. Let's talk about the details, starting with an example using **repeat**.

Let's say you are editing a file and you want to delete from the current position, backwards, to the beginning of the sentence. The key sequence to perform this operation is **C-x BS** (<Ctrl-X> <Backspace>). You type **C-x BS** to delete a sentence, and then decide you want to delete another one in the same way. You can, of course, type **C-x BS** again. However, it is better to use **C-x z** for two reasons.

First, **C-x BS** work only in this particular case. However, since **C-x z** is the universal way to repeat the last command, it always works.

Second — and this is important — **C-x z** allows you to use a particularly handy shortcut. Once you press **C-x z** once, you can repeat it again, as many times as you want, simply by pressing the letter **z**. For example, let's say you want to use **C-x BS** to delete five sentences. It would look like this:

C-x BS C-x z z z

Let me analyze the key sequences for you:

1. **C-x BS** deletes the previous sentence.
2. **C-x z** repeats the command.
3. **z** repeats the command a third time.
4. **z** repeats the command a fourth time.
5. **z** repeats the command a fifth time.

Although this may seem complicated the first time you see it, using **C-x z** in this way is fast and easy. Once you get used to it, your fingers will know what to do automatically without your mind having to think about it.

Moving on to **undo**: In Section 7.2, I mentioned that if you make a mistake, you can reverse it by pressing **C-x u** to invoke the **undo** command. As with the **repeat** command, you can press **C-x u** more than once to reverse more than one mistake. However, in addition to **C-x u**, there are two other key sequences that are also bound to the **undo** command: **C-/** (<Ctrl> -slash) and **C-_** (<Ctrl> -underscore). (For a discussion of commands and key bindings, see Section 6.1.)

C-x u is a longer, slightly more complex key sequence than **C-/** and **C-_**. However, there is an advantage to using it. In my experience, **C-x u** will always work, while **C-/** and **C-_** will sometimes not work with particular terminals and keyboards. My suggestion is to take a moment right now, and try all three of these key sequences on your system. See which ones work for you, and pick the one you like best. (My personal preference is **C-/**.)

Before we continue, I should point out that you will often see **C-_** referred to as **C--** (<Ctrl> -hyphen). This is because, on most keyboards, the (underscore) character is actually <Shift-hyphen> and the Shift key is ignored when you press <Ctrl> (see Section 4.2). Thus, **C-_** is actually the same as **C--**, and you will see it written both ways.

So why are there so many ways to use the **undo** command? Technically speaking, **C-/** is the primary key sequence bound to the **undo** command. However, **C-x u** is bound to **undo**, because it is easier for beginners to memorize. As you will find out, **C-x** is a common prefix, and it is easy to remember that **u** stands for "undo".

In addition, **C-_** is also bound to **undo** because, on some terminals, typing **C-/** actually sends the **C-_** character (and, as I just mentioned, **C-_** is the same as **C--**).

■ **Note**

When you are learning Emacs and you find something strange—for example, multiple key sequences bound to the same command—remember that there is always a good reason. You just need to find it.

Emacs has been around since 1975 (see Section 1.4), and nothing has happened by accident.

You can use **undo** repeatedly to reverse one change after another, moving backward in time, as many times as you want. This can be most useful, and combining it with **repeat** makes it even easier. Start by pressing **C-x u** to reverse the last change. Then press **C-x z** to repeat **undo** and reverse the next most recent change. You can now press **z** as many times as you want to reverse as many more changes as you want. For example, to reverse the last 10 changes to the buffer you would use:

C-x u C-x z z z z z z z z z

Or, using **C-/** instead of **C-X u**:

C-/ C-x z z z z z z z z z

Notice that the number of repeated **undo**s is equal to the number of **z**'s.[2]

Having talked about **undo**, I want to talk for a moment about "redo". Let's say that you are reversing multiple changes to the buffer, when you realize you have gone too far. Can you take back a few of the **undo**s? The answer is yes. If you are careful, you can use **undo** to reverse a previous **undo**. However, there is an important consideration.

When one **undo** follows another, Emacs assumes you want to reverse the previous change, going backwards in time. For example, if you type 10 **undo** commands in a row, Emacs will reverse the last 10 changes. However, the moment you type any other command, the chain is broken. Once this happens, if you type another **undo** command, it will reverse the last command, which was itself an **undo**. In this way, you can use a new **undo** as a "redo".

---

[2] This is what Paul McCartney was referring to when he wrote, in the last part of *Abbey Road*: "And in the end, the love you take is equal to the love you make."

I know that sounds confusing, so let's go slowly. Let's say you realize you have made a series of mistakes. You start using **undo** commands, one after another, to reverse the changes and go back in time. However, you accidentally go back too far. So, to break the **undo** chain, you purposely type a different command. You can now use another **undo** to go forward in time by reversing the previous **undo**.

When it comes to choosing a non-**undo** command, it is best to use something that doesn't make any changes to the buffer. A good choice is something like **C-f**, because all it does is move the cursor forward one position.

To show you how it works, here is an example. Look at it carefully, and then I'll explain:

C-/ C-x z z z z z z z z z C-f C-/ C-/ C-/ C-/

To start, I typed an **undo** command:

1. **C-/**

Then, I repeated this command 9 times:

2. **C-x z z z z z z z z z**

Next, I typed **C-f** to terminate the chain of **undo** commands:

3. **C-f**

Finally, I used **undo** 4 more times to reverse the previous 4 **undo**s:

4. **C-/ C-/ C-/ C-/**

If you want to make this even more obfuscated, remember that, after the **C-f**, you can use **C-x z** to repeat the new series of **undo** commands. So the following series of commands is equivalent to the one above:

C-/ C-x z z z z z z z z z C-f C-/ C-x z z z

Before we move on, please take a moment to re-read the last line and realize that it actually makes sense to you. Remember that I told you in Section 1.1 that using Emacs will make changes to the cells in your brain.

Sometimes, when we reverse an **undo**, we call it a "redo". Emacs doesn't have a redo command. However, as you can see, by using **undo** commands, you simulate a redo by typing a neutral command (in our case, **C-f**) followed by another **undo** command.

I must warn you that it will take a bit of practice to learn how to redo smoothly and successfully. This is because, behind the scenes, Emacs is doing things you will not understand at first, and it is easy to get confused. For example, if, after you end the chain of **undo** commands, you accidentally press the wrong key, another **undo** will reverse the accidental key, not the previous **undo**.

Rest assured, it all makes sense and it will come with practice, so be patient.

Section 7.4: The keyboard-quit Command (C-g)

One of the most important Emacs key sequences is **C-g** which is bound to the **keyboard-quit** command. You use **C-g** to cancel a partially typed command, a command that is already running, or a program that you started from within Emacs. We have talked about **C-g** earlier but, because it's so useful, I want to take a moment to focus on it.

The **keyboard-quit** command (**C-g**) can help you in the following situations:

Problem #1: You are in the middle of typing a command when you change your mind. You decide that you would just as soon forget the whole thing.

Solution: Press **C-g** to cancel the command.

Problem #2: You have started a command and it is not doing what you want.

Solution: Press **C-g** to cancel the command. If that doesn't work, press **C-g** again.

Problem #3: Something is happening that you want to stop.

Solution: Press **C-g**. It not only cancels commands, it stops programs (such as Lisp functions) that are running within Emacs.

Section 7.5: Emacs for vi Users

If you have some Unix experience, it is likely that you have used the **vi** text editor, most likely its modern version, Vim. The **vi** editor has two distinct modes (ways of working): COMMAND MODE and INSERT MODE. Before you can type text, you must be in insert mode, and before you can type a command, you must be in command mode. Thus, as you use **vi**, you will often find yourself changing back and forth from one mode to another.

Compared to **vi**, Emacs is a MODE-LESS editor. Specifically, since Emacs doesn't have an insert mode, you can type text whenever you want. (In **vi** terms, we might say that Emacs is always in insert mode.) Thus, in one sense, Emacs is simpler to use than **vi**, because you never have to change from one mode to another.

However, this simplicity does not come for free. In **vi**, the names of the commands are simple, and easy to type and remember. For example, to delete from the cursor to the end of the line, you type **D**. To save (write) your text to a file, you type **:w**. To search for a pattern using a regular expression (see Section 10.7), you press **/** (slash) and then type the expression.

In Emacs, all the commands require special keys like <Ctrl> and <Meta>, so they will not be confused with the letters, numbers and punctuation you might be typing. Thus, Emacs commands tend to look strange and can take longer to memorize. For

example, to delete from the cursor to the end of the line, you type **C-k**. To save your text to a file, you type **C-x C-s**. To search for a pattern using a regular expression, you type **M-C-s** (<Meta-Ctrl-S>) and then type the expression.

Is all this complexity worth it? I think so. All you need to do is memorize 40 to 50 basic Emacs commands (which is a lot easier than you might think), and you will be as comfortable as a brother-in-law living in the spare room. Moreover, since you won't have to expend so much mental effort switching back and forth from one mode to another, a part of your brain is freed up to think about other things (such as remembering all the key sequences).

In addition, there is another important difference between **vi** and Emacs. When you start using **vi**, your main goal is to memorize and learn how to use the various commands. With Emacs, you also need to learn how to use the commands, but there is more. Emacs is designed — on purpose — so that a lot of the time, you must understand what it is doing and why.

Depending on your point of view, this is either a virtue or a flaw. However, with understanding comes acceptance. You are embarking on what will become one of the longest relationships of your life.

Section 7.6: Commands to Control Windows

One of the tricks to being an Emacs virtuoso (or, at least, looking like an Emacs virtuoso) is to become a whiz at manipulating windows. So if you like to impress other people, you will find this section particularly useful and interesting.

When you are editing a single buffer, Emacs puts it in one large window. Thus, much of the time, you will be working with one buffer and only one window.

At various times, however, Emacs will create another window. This will happen automatically whenever Emacs has some information it needs to display. For example, when you start the Help facility (explained in Section 12.3), Emacs will create a new window and, within that window, display a buffer named ***Help***.

In addition, you can create a new window for yourself whenever you want. You can use the new window to display the same buffer as the old window or a completely different buffer. Remember, at any particular time, one window is designated as the selected window (see Section 6.3). This is the window that contains the cursor.

The commands to work with windows are shown in Figure 7-2. With a little practice, you will be zipping around like a greased snipe on his way home from Starbucks, moving from one window to another, manipulating windows like nobody's business.

Command	Description
C-x 0	Delete the selected window
C-x 1	Delete all windows except selected window
C-x 2	Split selected window vertically
C-x 3	Split selected window horizontally
C-x o	Move cursor to the next (other) window
C-x }	Make selected window wider
C-x {	Make selected window narrower
C-x ^	Make selected window larger
M-x shrink-window	Make selected window smaller

FIGURE 7-2. Commands for controlling windows.

Here is a good way to practice. Start Emacs using one of the following commands:

```
emacs -Q
emacs -Q -nw
```

You will have one large window with a ***scratch*** buffer. Now type **C-h b** to start the Help facility and display a list of all the key bindings. You now have two windows that you can use to practice the commands in Figure 7-2. When you are finished, you can stop Emacs by pressing **C-x C-c**.

The best way to become comfortable controlling windows is to spend time experimenting with the commands. At first, you will feel a little awkward, but it will soon become second nature to move from one window to another, delete a window, create a new one, and so on.

Please be sure you understand the difference between windows and buffers. As I explained in Section 6.2, a buffer is a work area and you can have as many as you want. A window is simply an area on your screen that Emacs uses to display the contents of a buffer. Thus, when you delete a window, you are *not* deleting the buffer that is displayed in that window. Any buffers that are not currently displayed are maintained invisibly in the background. Thus, you can have many buffers, only some of which are actually displayed in windows at the current time.

Before we leave this topic, there are a few points I would like to make about the commands in Figure 7-2. First, you will notice that there are two key sequences whose names might be a tad confusing. The "delete selected window" command is **C-x 0** (the number zero). The "move cursor to next window" command is **C-x o** (the lowercase letter "o"). It may help if you think of the letter **o** as standing for "other window".

Strictly speaking, **C-x o** (the letter "o") moves to what is called the NEXT WINDOW. When you have more than one window on your screen, Emacs moves from one to another in a specific order. If you have only two windows, the next window is simply the other window. If you have more than two windows, Emacs cycles from one to another, going from left to right and from up to down.

If you want to check this out for yourself, try the following experiment. Start Emacs:

```
emacs -Q
emacs -Q -nw
```

You now have one large window containing an empty buffer named **`*scratch*`**. Now press **`C-x 2`** and split the window into two windows, one on top of the other. Press **`C-x o`** a few times and watch how the cursor moves from one window to another.

Now press **`C-x 3`** and split one of the windows into two side-by-side windows. Again, press **`C-x o`** a few times and see how Emacs cycles through the three windows. Try using **`C-x 2`** and **`C-x 3`** to create some more windows and, each time, watch how **`C-x o`** moves the cursor. When you are finished, press **`C-x C-c`** to quit Emacs.

Now, take a look back at Figure 7-2. Notice that the command to make the selected window smaller (**`shrink-window`**) is not bound to a specific key sequence. Thus, you must execute this command explicitly by using **`M-x`**. Normally, though, you won't need the **`shrink-window`** command because whenever you make a window larger (by using **`C-x ^`**), Emacs automatically makes the other windows smaller. Thus, you can get by just fine simply by making a specific window larger as the need arises and letting Emacs adjust the other windows as it sees fit. Indeed, this is what most people do most of the time, which is why the **`shrink-window`** command does not really need its own key sequence.

Note

You will notice that commands to delete and split windows are similar: **`C-x`** followed by a number (**`0`**, **`1`**, **`2`** or **`3`**). It looks as if there might a pattern.

There isn't, so don't bother trying to invent one. Just practice and, after a few days, you will find that each individual command will become familiar on its own.

Section 7.7: Commands to Control Buffers

As we discussed in Section 6.2, a buffer is a work area that is maintained for you by Emacs. You can have as many buffers as you want at the same time, each with its own name. At all times, you will have at least one buffer. If you start Emacs without specifying the name of a specific file, Emacs will create a buffer named **`*scratch*`**.

Not all of your buffers need to be displayed in a window. Indeed, it is common to have an assortment of buffers to use as separate work areas, of which only one or two are actually displayed in windows. As the need arises, you can change which buffers are displayed.

■ **Note**

It is important to realize that Emacs can handle as many buffers as you need, even though you may have only one or two windows. This means that, when you want to work with multiple files, you don't need to run a separate instance of Emacs for each file.

Once you master working with windows and buffers, you will be able to use a single instance of Emacs to edit as many files as you want at the same time, because it's so easy to switch back and forth from one buffer to another.

Figure 7-3 shows the commands you can use to control your buffers. These commands work together with the window-oriented commands I described in Section 7.6 (see Figure 7-2).

Command	Description
C-x b	Display a different buffer in selected window
C-x b	Create a new buffer in selected window
C-x C-b	Display a list of all your buffers
C-x k	Kill (delete) a buffer
C-x 4 b	Display a different buffer in next window
C-x 4 C-o	Same as **C-x 4 b**, but don't change selected window

FIGURE 7-3. Commands for controlling buffers.

The most important of these commands is **C-x b**. You use this command to tell Emacs that you want to work with another buffer. This new buffer will be displayed in the window in which you are currently working (the selected window). The buffer that is currently in the window will be replaced, but not lost.

When you press **C-x b**, Emacs will wait for you to enter the name of the buffer with which you want to work. If this buffer already exists, Emacs will just move it into the window. Otherwise, Emacs will create a brand new empty buffer in the window using the name you specified. Thus, **C-x b** is the command to use when you want to create a new buffer.

If you have more than a few buffers, it is easy to forget their names. To remind yourself, you can press **C-x C-b** to display a list. When you do, you will notice that Emacs has created a new buffer called ***Buffer List*** to hold the list itself.

I mentioned that when you change the contents of a window, the buffer that was replaced is not destroyed: it exists in the background and you can recall it whenever you want. However, from time to time, you may actually want to delete a buffer. To do so, use the **C-x k** (kill) command. When you do, Emacs will wait for you to enter the name of the buffer you want to kill. The default will be whatever buffer is in the selected window. If this is the buffer you want to delete, just press <Enter>. Otherwise, type the name of another buffer and press <Enter>. If you decide to cancel the command, press **C-g**, the Emacs cancel key.

Note You will not be allowed to delete all of your buffers. At the very least, Emacs will force you to have a single buffer named **`*scratch*`**.

There will be many times when you are working with one buffer and you want to display another buffer in a different window. To do so, there are two commands you can use. The command **`C-x 4 b`** tells Emacs to display whichever buffer you specify in a different window. This new window then becomes the selected window. Thus, the **`C-x 4 b`** command allows you to switch to another buffer while still being able to see the contents of the old buffer. As with **`C-x b`**, if the buffer does not already exist, the **`C-x 4 b`** command will create a new buffer for you.

Sometimes you will want to look at the contents of another buffer without changing the selected buffer. For example, say that you are typing a letter to someone and you need to display his or her address which is in a different buffer. You will want to display the contents of this second buffer without moving away from the window in which you are typing.

In such cases, use the **`C-x 4 C-o`** command. This is similar to the **`C-x 4 b`** command, except that the selected window does not change. One restriction (which only makes sense) is that you must specify the name of a buffer that already exists. After all, there is no point in displaying an empty buffer in another window.

Note

Most Emacs commands act on the selected buffer. However, commands that begin with **`C-x 4`** act on another buffer. Learning how to use the **`C-x 4`** commands allows you to control your buffers smoothly and quickly.

There are also **`C-x 4`** commands that deal with files. These commands are explained in the Section 7.8.

Section 7.8: Commands for Working With Files

The crucial thing to understand about files is how they relate to buffers. In Section 2.15 and Section 2.16, we discussed the Unix file system and talked about the technical definition of a file. In this section, we can assume that a file is a collection of information that is given a name and stored on a device (usually a disk).

In Section 5.1, I explained that when you start Emacs, you can specify the names of one or more files. If you do this, Emacs will automatically read the contents of each file into its own buffer. Once you have a lot of experience, you will find that it is better to start Emacs without file names. When you want to start working with a file (which may be right away), you use a command that tells Emacs to read the contents of the file into a buffer.

■ **Note**

Please remember: a buffer is a *temporary* work area. As such, it disappears when you quit Emacs. Thus, if you want to save the contents of a buffer to a file, you must do so before you quit.

For the same reason, as you are working, it behooves you to save your buffers regularly.

The commands to work with files are summarized in Figure 7-4. Most of the commands act on the selected window (the window in which the cursor resides.) However, the commands that begin with **C-x 4** act on the next window. This makes it easy to load a file into another window without moving from your current window. (We discussed the idea of the next window in Section 7.7.)

Command	Description
C-x C-f	Switch to buffer containing specified file
C-x C-v	Replace buffer contents with specified file
C-x C-s	Save a buffer to file
C-x C-w	Save a buffer to specified file
C-x i	Insert contents of a file into buffer
C-x 4 C-f	Read contents of file into next window
C-x 4 f	Same as **C-x 4 C-f**
C-x 4 r	Same as **C-x 4 C-f**, read-only

FIGURE 7-4. Commands for working with files.

If this is new to you, I realize that the commands in Figure 7-4 may look a bit confusing. To help you, here is some quick advice: the only file commands you really need to memorize are **C-x C-f** to read a file and **C-x C-w** to save a file. (Think of **f** as standing for "file" and **w** as standing for "write".)

The **C-x C-f** command tells Emacs to read the contents of a file into a buffer. Emacs will use the name of the file as the name of the buffer. If the file does not already exist, Emacs will create a new buffer by that name. When Emacs copies the contents of a file into a buffer, we say that you VISIT the file.

The idea of visiting a file is important because it implies an association between a buffer and a file. When you visit a file, Emacs remembers which buffer is associated with that particular file. Once you make changes to that buffer, Emacs will not let you quit without giving you a chance to save the contents of the buffer back to the file. The idea is to make it difficult to lose your work by accident.

However, when you create a new buffer that is not tied to a particular file, Emacs will be more than glad to let you quit without reminding you to save your work. (Read that last sentence again.)

Here is an example. You have two windows. In the first window, you have used the **C-x C-f** command to read in a file named **griffin** that contains a letter to a friend in the South Seas. When you entered this command, Emacs created a buffer named **griffin** in which to copy the file. It is this buffer that you are looking at in the window. In the second window, you have used the **C-x b** command to create a brand new buffer named **sabine**, into which you have typed a letter to a friend in England.

Now, both windows are similar in that they each contain a buffer that holds some text. However, in the first window, a file is being visited, while in the second window no file is being visited. Thus, if you were to quit Emacs, you would be asked if you want to save the contents of the **griffin** buffer back to the file, but you would *not* be asked if you want to save the **sabine** buffer.

Now, let's say you want to start working with a new buffer. You have two ways to create that buffer. You can use **C-x b** or you can use **C-x C-f**. (Remember, each of these commands will create a new buffer if the one you specify does not already exist.) The difference is that if you use **C-x b**, Emacs will create a buffer that is not tied to any particular file. If you use **C-x C-f**, the new buffer will be associated with a file of the same name. (That is, you will be visiting that file.) Thus, when you quit, you will be asked if you want to save the contents of the buffer to a file.

Note When you want to create a new buffer for work you do *not* want to save, use the **C-x b** command. When you want to create a new buffer for work you do want to save, use the **C-x C-f** command.

The **C-x C-v** command copies the contents of a file into the current buffer, replacing the current contents of that buffer. You use **C-x C-v** when you want to switch to a new file and you don't mind losing what you are working on.

When you replace the contents of a buffer using **C-x C-v**, whatever was in the buffer will be deleted. For your own protection, Emacs will ask you for confirmation before it replaces a buffer that has not been saved.

If you want to insert the contents of a file into the current buffer without losing what is already in the buffer, use the **C-x i** command. The contents of the file you specify will be copied into the buffer at the current cursor position. The original contents of the buffer will be moved to make room for the new data, but will not be deleted.

There are two commands you can use to save the contents of a buffer to a file. Use the **C-x C-s** command when you are visiting a file and you want to save the contents of the buffer back to that same file. Use **C-x C-w** when you want to save a buffer to a different file. For example, if you are editing a file named **griffin**, the **C-x C-s** command will copy the current contents of the **griffin** buffer to the file named **griffin**. Obviously, this is a command you should use frequently to save your work.

Whenever you need to specify a file name, Emacs will help you by displaying the name of the current directory in the minibuffer. You can then type the name of the file you want. If the beginning of the file name is unique, you can save keystrokes by using the completion facility we discussed in Section 6.7.

Here is an example using a Unix file system (see Section 2.17). Let's say that your current directory is named **memos**. This directory lies within your home directory. When you press **C-x C-f**, Emacs will display the following prompt in the minibuffer:

```
Find file: ~/memos/
```

The **~** character (explained in Section 2.17) is an abbreviation for your home directory.

This particular prompt tells you that Emacs is guessing that you want a file in this particular directory. If that is the case, type the name of the file and press <Enter>. If you want a file in a different directory, you can use the Backspace key to erase the directory name and specify your own. At the end of this new directory name, type a **/** (slash), then the file name, and then press <Enter>.

As I explained in Section 7.7, key sequences that begin with **C-x 4** are used to manipulate the next window. The **C-x 4 C-f** command works like **C-x C-f** except that it acts on the next window. As a convenience, you can use **C-x 4 f** instead of **C-x 4 C-f**.

The **C-x 4 r** command is similar except that it sets the buffer to be read-only when it reads in the file. This allows you to examine an important file in another window without having to worry about changing the file by accident.

CHAPTER 8

The Cursor; Line Numbers; Point and Mark; The Region

Section 8.1: The Cursor and the Idea of Point

Emacs has a special name for the current position of the cursor. This location, within the buffer, is called POINT (not "the point", just "point".) The idea of point is important because it is at this location that whatever you type is inserted into the buffer. When you read Emacs documentation and the descriptions in the Help facility, you will see many references to point.

Although the cursor is under or on a particular character, point is actually between two characters: the one at the cursor position and the character immediately to its left. For example, say that, in your buffer, you are reading the word **`tergiversate`** and the cursor is on the **`g`**. Point is considered to be between the **`r`** and the **`g`**.

What's in a Name?

Point

The original Emacs was developed to be a set of editing macros for an obtuse text editing facility named TECO. Within TECO, you used the `.` (period) character as the command for accessing the current location within the text. Since the `.` character was really just a dot, the command that it represented was referred to as the "point" command. In Emacs, the current location within the current buffer is marked by the cursor, and is referred to as "point".

It is important to realize that each buffer has its own point which is carefully maintained by Emacs. Of course, there is only one cursor, and it is used to show where point is in the buffer that is currently active. However, Emacs remembers where point is in each buffer so that, as you switch from one buffer to another, Emacs knows exactly where to place the cursor.

H. Hahn, *Harley Hahn's Emacs Field Guide*, DOI 10.1007/978-1-4842-1703-0_8

Section 8.2: Moving the Cursor

Moving the cursor is straightforward. You can move it up and down, and backward and forward. The commands for moving the cursor are summarized in Figure 8-1.

Backward	Forward	
C-b	**C-f**	a single character
<Left>	<Right>	a single character
M-b	**M-f**	a word
C-p	**C-n**	a line
<Up>	<Down>	a line
M-a	**M-e**	a sentence
M-{	**M-}**	a paragraph
Beginning	End	
C-a	**C-e**	the current line
M-<	**M->**	the entire buffer

FIGURE 8-1. Commands for moving the cursor.

Notice that there are two types of commands. First, there are commands that move forward or backward a specific amount. For example, you can move to the left or right by a single character by using **C-b** and **C-f** respectively. There are similar commands to move by a single word, line, sentence or paragraph.

As a convenience, you can use the cursor control keys to move a single position at a time. Thus, pressing the Left key is the same as **C-b**, and pressing the Up key is the same as **C-p**.

Second, there are commands that move to the beginning or end of something. You will find these commands to be especially useful. For example, to jump to the very beginning of the buffer, use **M-<**(<Meta> -less-than sign). To jump to the end of the buffer, use **M->**(<Meta> -greater-than sign).

If you look at the key sequences, you can see some patterns. For example, the letters **b** and **f** stand for "backward" and "forward"; the letters **p** and **n** stand for "previous" and "next"; and so on. Still, you don't really need to memorize Emacs keys in this way. Just practice for a few days, and you will remember without even trying.

Section 8.3: Text Modes; Paragraphs and Sentences

In Section 8.2, we talked about the commands to move the cursor through a paragraph or a sentence. You can see these commands in Figure 8-2:

Command	Description
M-}	Move forward one paragraph
M-{	Move backward one paragraph
M-e	Move forward one sentence
M-a	Move backward one sentence

FIGURE 8-2. Commands for moving the cursor through a paragraph or a sentence.

To use these commands, you need to know exactly how Emacs defines a "paragraph" and a "sentence", so let's talk about that.

The exact definition of a paragraph depends upon which "major mode" you are using. We will discuss major modes in Section 11.2. For now, I'll just say that the major mode controls how Emacs behaves as you are editing. When you are writing ordinary text in English (or another language), the major modes you are most likely to use are listed in Figure 8-3.

Mode	Command
Fundamental mode	**fundamental-mode**
Text mode	**text-mode**
Indented Text mode	**indented-text-mode**
Paragraph-Indent Text mode	**paragraph-indent-text-mode**

FIGURE 8-3. Major modes to use when editing a human language.

To change from one mode to another, type **M-x** (see Section 6.1) followed by the name of the mode, then press <Enter>:

```
M-x fundamental-mode
M-x text-mode
M-x indented-text-mode
M-x paragraph-indent-text-mode
```

Fundamental mode is the generic Emacs major mode. It is the default Emacs major mode, the one you use when you don't have a reason to use another mode. If you are writing English prose, you are better off with Text mode or Paragraph-Indent Text mode.

Text mode is for writing in a human language (as opposed to, say, a computer program). Use this mode when your paragraphs are separated by blank lines.

Paragraph-Indent Text mode is similar to text mode. Use this mode when the beginning of each paragraph is indented.

Indented Text mode is for when you are working on something that has a lot of indentation and you don't care much about paragraphs, for example, when you are creating an outline.

At this point, I am almost ready to give you the technical definition of a paragraph. I just need to define three more technical terms. (As the medieval Persian poet Saadi Shirazi (1210-1291) once said when *he* was learning Emacs, "Have patience. All things are difficult before they become easy.")

First, WHITESPACE refers to any combination of **SPC**, **TAB** or **RET**. (These characters are described in Section 4.4.) It is called whitespace because when you print on paper, it looks white.

Second, a BLANK LINE is a line that is either empty or that contains only whitespace.

Finally, an INDENTATION occurs when a line starts with one or more tabs or spaces.

With Fundamental mode, Text mode, and Indented Text mode, you indicate your paragraphs by putting a blank line between them. With Paragraph-Indent Text mode, you indicate your paragraphs by indenting them, by putting a blank line between them, or both. So here is the definition:

A PARAGRAPH is a sequence of characters that is separated from other characters by a blank line or (with Paragraph-Indent Text mode) that is indented.

Now that you know all that, take some time to experiment. Open a new buffer and type various types of text. Then use the **M-x** commands above to switch from one major mode to another and see what happens when you type **M-}** and **M-{**.

To finish, a SENTENCE is a sequence of characters that ends with a **.** (period), **?** (question mark), or **!** (exclamation mark) followed by two spaces. A sentence also begins or ends wherever a paragraph begins or ends. To make sure you understand this, open a buffer with some text, and see what happens when you use **M-a** and **M-e**.

Section 8.4: Repeating a Command: Prefix Arguments

To perform a command a specified number of times, you type what is called a PREFIX ARGUMENT in front of the command. For example, you might type the prefix argument that means "repeat the following command 6 times" and then press **C-p**. This will move the cursor up 6 lines. Because the prefix argument specifies a number, it is also referred to as a NUMERIC ARGUMENT. (The term ARGUMENT is a programming word that describes a value passed to a program when it is executed.)

To specify a prefix argument, hold down the Meta key and type a number. For example, to move the cursor up 6 lines, press **M-6 C-p**. An alternative is to use <Esc> instead of <Meta>. Thus, to move 15 characters to the right, you could press **ESC 15 C-f**.

You can use a prefix argument with any command, and Emacs will interpret the numeric value in the way that makes the most sense for that command. Where prefix arguments really come in handy is when you combine them with the cursor movement commands we discussed in Section 8.2.

■ **Note** Take a few moments and practice various combinations of prefix arguments with the cursor movement commands in Figure 8-1.

The number you specify can be either positive or negative. If you use a negative number with a cursor movement command, it will move in the opposite direction. For example, to move the cursor up 6 lines, use either **M-6 C-p** or **M--6 C-n**. (Perhaps this last command would be clearer if I wrote it as **ESC -6 C-n**.)

If you use the Meta key, you will hold it down as you type a prefix argument. For example, when you use the **M-6 C-p** command, you will have to hold down the <Meta> key as you type the **6**. Some people find this inconvenient, so there are two alternatives. First, as I mentioned, you can press **ESC** instead of holding down <Meta>. Second, you can press **C-u** instead. (The name comes from the fact that this key is bound to the command **universal-argument**.) Thus, to move the cursor down 6 lines you can use either **M-6 C-n** or **C-u 6 C-n**. Although this looks like an extra keystroke, it is actually easier, especially when you are using a multi-digit prefix argument. For example, to move the cursor down 120 lines, you can use **C-u 120 C-n**.

As a final shortcut, the **C-u** command has a special meaning when you use it before a command *without* specifying a number. In such cases, the **C-u** key tells Emacs to repeat the next command 4 times. And you can type more than one **C-u** in a row to multiply this effect. This may seem a bit strange at first, but it is really, really useful, so take a moment to figure it out.

Here is a simple example. To move the cursor down 4 lines, you can use **M-4 C-n** or **C-u 4 C-n**. But, as a shortcut, you can use **C-u** by itself: **C-u C-n**. Similarly, to move the cursor 4 characters to the left, you can use **C-u C-b**.

The power of the **C-u** prefix comes when you use more than one in a row. Because each such prefix multiplies the next command by a factor of 4, using two in a row tells Emacs to repeat a command 16 times. Using three in a row repeats a command 64 times.

For example, to move the cursor down 16 lines, you can use **C-u C-u C-n**. To move the cursor 64 characters to the right, you can use **C-u C-u C-u C-f**. Although this may seem a bit awkward, it is actually quick and easy to type.

Once you get the hang of it, you can use more than one series of commands to combine movements. For, example, let's say you want to move the cursor down 63 lines. It is actually very fast — and totally cool — to move down 64 lines and then up one:

```
C-u C-u C-u C-n C-p
```

Try it and see if I'm not right.

To summarize, the various prefix argument combinations are shown in Figure 8-4.

Prefix	Effect
M-*number*	Repeat command specified number of times
ESC *number*	Repeat command specified number of times
C-u *number*	Repeat command specified number of times
C-u	Repeat command 4 times
C-u C-u	Repeat command 16 times
C-u C-u C-u	Repeat command 64 times
C-u C-u C-u C-u	Repeat command 256 times

FIGURE 8-4. Prefix argument combinations.

■ **Note**

The **C-u C-u** prefix is especially useful when you are working with characters and lines. This is because 16 characters are about one fifth of a line and 16 lines are about one third of a standard-sized screen. It is worth a few moments of your time to practice putting together the **C-u C-u** prefix with your favorite character and line commands, just to fix them firmly in your mind. For example, try:

C-u C-u C-f (move 16 lines to the right)
C-u C-u C-b (move 16 lines to the left)
C-u C-u C-n (move 16 lines down)
C-u C-u C-p (move 16 lines up)

You may find it handy to hardwire these particular combinations directly into your motor cortex (the lump of gray matter in your prefrontal gyrus, just anterior to the central sulcus). The details for doing so, however, are beyond the scope of this book.

Section: 8.5: Moving Through the Buffer

There will be many times when you want to page through the buffer. Perhaps the most common example is wanting to read something from beginning to end. You start at the top of the buffer and read one screenful at a time.

When you tell Emacs to display information that is just beyond the border of your window, we say that you are SCROLLING. For example, if you have read what is on the screen and you move down to the next screenful, we say that you scroll down. Similarly, you can display the previous screenful by scrolling up.

Notice that we talk about scrolling as if *you* do it, rather than as if Emacs is doing it. For example, you might tell a friend, "To find the secret phone number, open the file I sent you and scroll down until you see the entry for Kalle Anka." This is a common way of speaking that reminds us that computers are actually extensions of our minds.

Figure 8-5 shows the commands you can use to move throughout the buffer. The scrolling commands are completely straightforward, and there is not much I want to say about them, other than be sure to memorize **`C-v`** and **`M-v`** *this very minute*. These are two crucial commands that you will use every day of your life, so don't even leave this paragraph without committing them to memory. The right and left scrolling commands are less important: you need them only when you are dealing with unusually long lines.

Command	Description
`C-v`	Scroll down one screenful
<PageDown>	Same as **`C-v`**
`M-v`	Scroll up one screenful
<PageUp>	Same as **`M-v`**
`M-C-v`	Scroll down in the next window
`M-<`	Jump to the beginning of buffer
`M->`	Jump to the end of buffer
`C-l`	Redisplay the screen, current line in middle

FIGURE 8-5. Commands to move throughout the buffer.

Note If you are using a terminal window, and typing **`M-v`** (<Alt-V>) pulls down a menu from the top of the window, it is because your terminal window program is interpreting <Alt> as a menu key. For instructions on how to fix this problem, see Section 4.6.

One variation on the scrolling commands is **M-C-v**. This command scrolls down in the next window. (We discussed the idea of the next window in Section 7.6.) Using **M-C-v**, you can scroll through the next window without having to leave the window in which you are working. Unfortunately, there is no easy way to scroll *up* in the next window.

For completeness, I have included the **M-<** (jump to the beginning of the buffer) and **M->** (jump to the end of the buffer) commands in Figure 8-5. I described these commands in Section 8.2, when we discussed moving the cursor. However, these commands also belong here because they are handy for zipping around.

Finally, there is the **C-l** (lowercase letter "L") command. This command redisplays the screen so that the line on which the cursor lies is in the middle of the screen. This command is handy when the cursor is near the bottom of the screen and you want to pull it up somewhat to read the lines underneath. **C-l** is a useful command that is all too often neglected. Take a moment and try it for yourself to see how useful it can be.

Section 8.6: Using Line Numbers

Most of the time, you will move around the buffer in small and large jumps or by searching for a specific pattern (as we will discuss in Section 10.1). However, there will be times when it is handy to be able to jump directly to a specific line based on its position in the buffer. For example, you may want to jump to line 43. There are two commands you can use in this regard. They are described in Figure 8-6.

Command	Description
M-g g	Jump to line with specified number
M-x line-number-mode	ON/OFF: display line number on mode line

FIGURE 8-6. Commands to use line numbers.

To help you orient yourself within the buffer, Emacs displays the current line number on the mode line. You can see an example in Figure 8-7. In this case, we are looking at a buffer named ***Help***. The cursor is currently on line 43, and the top line of the screen is 59 percent of the way through the buffer. (The top line of the buffer is considered to be line 1.)

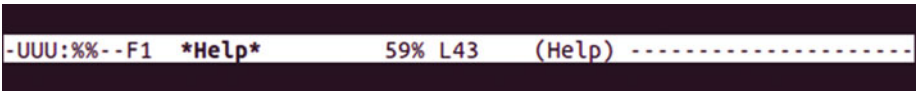

FIGURE 8-7. The mode line showing the current line number.

Emacs displays the current line number on the mode line. In this case, the cursor is on line 43, and the top line of the screen is 59 percent of the way through the buffer.

If the current line number is not displayed on the mode line, you can display it by typing:

M-x line-number-mode

This command acts like a toggle, turning the line number on and off.

To jump to a particular line, type **M-g g**. Then type the line number and press <Enter>, and the cursor will jump to the line you specify. For example, to jump to line 43, type:

M-g g 43 <Enter>

Section 8.7: Mark, Point, and the Region

Within Emacs there are commands that operate on various character groupings. For example, you can work with single characters, words, lines, sentences and paragraphs. To provide flexibility, Emacs also lets you define an area of the buffer — called a REGION — which can be as long or short as you want. Once you define a region, you can operate on it using any one of several commands. For example, you can define a region of, say, thirteen and a half lines, and then erase it. Or you can define a region of ten words, and then change them all to uppercase.

Here is how it works. A region is defined as all the characters between two locations: MARK and point. Point you already know; it is the location of the cursor. Mark is a location you can set for yourself.

There are several ways to define mark. The simplest way is to move the cursor to wherever you want mark to be and then type **C-SPC** (<Ctrl-Space>). Mark is now at that location. You can then move the cursor to a new location (which becomes point). The region is now all the characters between mark and point.

The best way to understand mark and point is to think of them as two locations in the buffer, both of which you can set. However, since there is only one cursor, Emacs can only show you one of these locations (point); you will have to remember where mark is. Still, it's not all that hard once you get used to it. Most of the time you will set mark for a particular purpose, and then use it right away before you forget where it is. A simple example will make it all clear.

In order for you to understand the example I am about to show you, I will tell you that the command **M-=** counts the number of lines, words and characters in a region. Let's say you are writing an essay about the psychology of investing,[1] and you

[1] **www.harley.com/money-and-economics/how-thinking-affects-investing/**

have just typed the following five lines into an empty buffer. You want to know how many words you have written.

```
To invest well requires us to develop our ability to think
well and make decisions, even in difficult situations. Doing
so, however, is difficult because our feelings and our
intuition can get in the way, often without our knowing what
is really happening.
```

The plan is to define the region to consist of these five lines, and then use **M-=** to count the words.

To start, move the cursor to the beginning of the buffer (**M-<**) and type **C-SPC**. This sets mark to be at this location. Next, move the cursor to the end of the buffer (**M->**). Now mark is at the beginning of the first line, and point is at the end of the last line. Type **M-=** to count the words. In the echo area (the bottom line of the screen; see Section 6.5), you will see:

```
Region has 5 lines, 43 words, and 247 characters.
```

These are important concepts, so I want to take a moment to be precise. As you may remember from our discussion in Section 8.1, point is not exactly the same as the cursor. The cursor sits under or on a particular character. Point is really between two characters, the one at the cursor and the one to its left. For example, consider the line:

```
abcdefghijklmnopqrstuvwxyz
```

If the cursor were on the **m**, point would be between the **l** and the **m**. When you set mark, it works the same way. Say that while the cursor is on the **m**, you press **C-SPC**. Mark (and point) are now both between the **l** and the **m**. Now, you move the cursor to be on the **g**. Mark is still between the **l** and the **m**, and point is now between the **f** and the **g**. Thus, the region consists of the letters **ghijkl**. If you were to press **M-=**, the echo line would display:

```
Region has 1 line, 1 word, and 6 characters.
```

■ **Note** When you are defining a region, be sure that the rightmost boundary is set to the character *after* the last one on which you want to operate.

Section 8.8: Using Mark and Point to Define the Region

A region is defined as all the contiguous characters between mark and point. Point is always at the position of the cursor. Thus, to define a region, all you need to do is set mark.

Broadly speaking, there are two ways in which mark can be set. First, as we discussed in Section 8.7, you can use a command that sets mark. Second, many commands that perform some function or other automatically set mark to a new value. When this happens, you will see a message in the echo area telling you that mark has been set.

For example, when you use a command that inserts text into the buffer, Emacs will finish the operation by setting mark at one end of the new text and point at the other end. Thus, the region will contain the newly inserted text.

Figure 8-8 shows the commands that explicitly set mark. Two of these commands set mark without changing point (**C-SPC** and **M-@**). Two other commands set both mark and point (**M-h** and **C-x h**). A final command interchanges the location of mark and point (**C-x C-x**).

Command	Description
C-@	Set mark to current location of point
C-SPC	Same as **C-@**
C-x C-x	Interchange mark and point
M-@	Set mark after next word (do not move point)
M-h	Put region around paragraph
C-x h	Put region around entire buffer

FIGURE 8-8. Commands to set mark and define a region.

Strictly speaking, the command to set mark is **C-@**. However, it happens that with many terminals, pressing **C-SPC** will generate the same character as **C-@**. Thus, although **C-SPC** is not a real character, we say that you can use it to set mark. When you do so, you are really using **C-@**, but **C-SPC** is a more convenient key sequence. If **C-SPC** does not work on your terminal — for example, if it generates a regular **SPC** character — you will have to use **C-@**. (On most keyboards, you use **C-@** by pressing **C-2**, because the **2** is the same key as **@**. You do not need to hold down the Shift key.)

Once you set mark, it stays where it is until you change it explicitly or until another command changes it. When you define a region, it does not matter whether mark comes before or after point. Nor does it matter which one you set first.

As I explained in Section 6.2 and Section 7.7, you can work with as many buffers as you want at one time. In the same way that each buffer has its own cursor, each buffer also has its own point and mark. Thus, if you set mark in one buffer and then move to another buffer to do some work, when you move back to the first buffer, the original mark will still be there.

Because mark is invisible, you may forget where it is. Unlike point, which is marked by the cursor, there is no way to look at the screen and see mark. In such cases, you can use the **C-x C-x** command to exchange the location of mark and point. Thus, the new location of the cursor will be where point was. To move the cursor back to its original location, simply press **C-x C-x** again.

■ **Note** Normally, you will set and use mark within a short time, so you will not forget its location. However, if you do, you can visualize the region by pressing **C-x C-x** twice. The cursor will jump back and forth from one boundary to the other.

You can always set mark by moving to wherever you want it and pressing **C-SPC** or **C-@**. However, there are three other commands that provide handy shortcuts.

The **M-@** command sets mark after the current word. For example, let's say that you are editing some text that contains the sentence:

```
Okay boys, let's defenestrate him.
```

The cursor is currently on the **d** in **defenestrate**. You press **M-@**. This sets mark to be at the space at the end of the word (after the **e**). Point will not change. You can verify this by pressing **C-x C-x**. This is a good way to set the region to contain a particular word on which you want to perform an operation.

If you want to set mark to be more than one word away, you can use a prefix argument (explained in Section 8.4). In our previous example, for instance, let's say that the cursor is once again on the **d** in **defenestrate**, and you want to set mark to be after the end of the word **him** (2 words away). Use **ESC 2 M-@**. This leaves mark at the period. To set mark to be 10 words away, use **ESC 10 M-@**, and so on.

The next command, **M-h**, sets the region by moving both mark and point to contain an entire paragraph. (Within Emacs, a "paragraph" starts with one or more **space** or **tab** characters, or is preceded by a blank line.) If the cursor is within a paragraph, **M-h** sets point to the beginning of the paragraph and mark to the end of the paragraph. Thus, when you press **M-h**, it not only sets the region, it also moves the cursor to the beginning of the paragraph. If you press **M-h** when the cursor is on a blank line, point and mark will be set to the beginning and end of the following paragraph.

The final command, **C-x h**, marks the entire buffer as being in the region. It does this by moving point to the beginning of the buffer and mark to the end of the buffer.

Section 8.9: Operating on the Region

In Section 8.7 and Section 8.8, we discussed how to set mark and point, and thereby define the region. The reason we do this is to make it easy to perform an operation on all the characters in the region. Figure 8-9 shows the Emacs commands you can use.

Command	Description
C-w	Kill (erase) all the characters
C-x C-l	Convert the characters to lowercase
C-x C-u	Convert the characters to uppercase
M-=	Count the lines and characters
M-|	Run a shell command, use the characters as data

FIGURE 8-9. Commands that act upon the region.

Generally speaking, the most useful of these commands is **C-w**. This command kills (erases) the entire region. If you change your mind after the deletion, you can type **C-x u** (or **C-/** or **C-_**) to undo the operation (see Section 7.3).

The **C-x C-l** and **C-x C-u** commands convert all the characters in the region to upper- and lowercase respectively. These commands work well with the mark-setting commands to change the case of a word or group of words.

For example, to change one word to uppercase, move the cursor to the beginning of the word, and press **M-@** (set mark at end of word) followed by **C-x C-u**. To change 5 words to lowercase, move to the beginning of the first word, and press **ESC 5 M-@** (set mark at end of fifth word) followed by **C-x C-l**.

Note When you use **C-x C-l** or **C-x C-u**, you may get a message that they are disabled. If so, read the discussion in Section 6.8 on how to enable such commands.

Here is one last example. Your cursor is in the middle of a line, and you want to change the entire line to uppercase. Type **C-a** (move to beginning of line), **C-SPC** (set mark), **C-e** (move to end of line), and finally, **C-x C-u** (change region to uppercase).

The **M-=** (Meta equals sign) command will count all the lines and characters in the region. This command is handy if you are a writer who has to keep measuring his output in order to convince his editor that he is making progress. (I am not mentioning any names here.) Combined with the region-defining commands, **M-=** works quickly and easily.

For example, say that you want to find out how many lines are in the buffer. The **C-x h** command will set the region to the entire buffer. Thus, to count all the lines in the buffer, all you need to type is **C-x h** and then **M-=**. Here is some typical output:

```
Region has 108 lines, 1724 characters
```

Try it: it's too cool for words.

Finally, the **M-|** (Meta-vertical bar) command will send the contents of the region to a shell command to be processed. To store the output, Emacs will create a buffer named ***Shell Command Output***. If this buffer already exists, its contents will be replaced by the output of the new command.

This command is incredibly useful, so let's look at a few examples. First, let's say you want to sort all the lines in the buffer. All you have to do is use the **C-x h** command to set the region to the entire buffer, and then use **M-|** to run the **sort** command. To test this out, let's create a customized list of all the Emacs commands in alphabetical order.

To get the raw material, we can use the built-in Help facility (described in Section 12.3. The command to use Help is **C-h**. If you type **C-h b**, Emacs will create a new buffer, named ***Help***, that contains descriptions of all the key bindings. Each line contains the name of the key, followed by the name of the command to which it is bound. Here are two examples:

```
C-h b    describe-bindings
ESC |    shell-command-on-region
```

(Notice that the <Meta> key is described as **ESC**.)

So all we have to do is generate a buffer full of key binding descriptions and sort them. When we do, we will use the Unix **sort** command with the **-u** (unique) option. This option eliminates all duplicate lines. In this case, the **-u** option will effectively eliminate all but one of the blank lines. So here is how to create your own alphabetical list of Emacs commands:

1. **C-h b**: Create a buffer named ***Help*** that contains the key descriptions.
2. **C-x o**: Change to the ***Help*** buffer.
3. **C-x 1**: Delete all windows except selected window.
4. **C-x h**: Set the region to be the entire buffer.
5. **M-| sort -u** <Enter>: Sort all the lines in the buffer.
6. **C-x o**: Change to the ***Shell Command Output*** buffer.

At this point, you may want to save the list to a file for future reference. If so, use the command:

7. **C-x C-w**: Save the buffer to a file.

Before we leave this topic, here is one more useful example. Unix has a **`fmt`** command you can use to format text. The **`fmt`** command makes your text look as uniform as possible, while preserving paragraphs and indentations. (In Emacs, this is known as "filling" text.) Using **`fmt`** is a nice way to smooth out ragged text that has been the victim of brutal modifications and editing.

To format the entire buffer, use **`C-x h`** (set the region), followed by **`M-| fmt`** (process the region with the **`fmt`** command).

Being able to use Unix commands from within Emacs is a particularly powerful tool. We will discuss this topic in detail in Section 12.1.

Note Emacs has a command, **`M-x fill-region`** that has much the same effect. This main difference is that **`M-x fill-region`** changes the original region. Using **`M-| fmt`** creates a new buffer containing only the formatted text.
