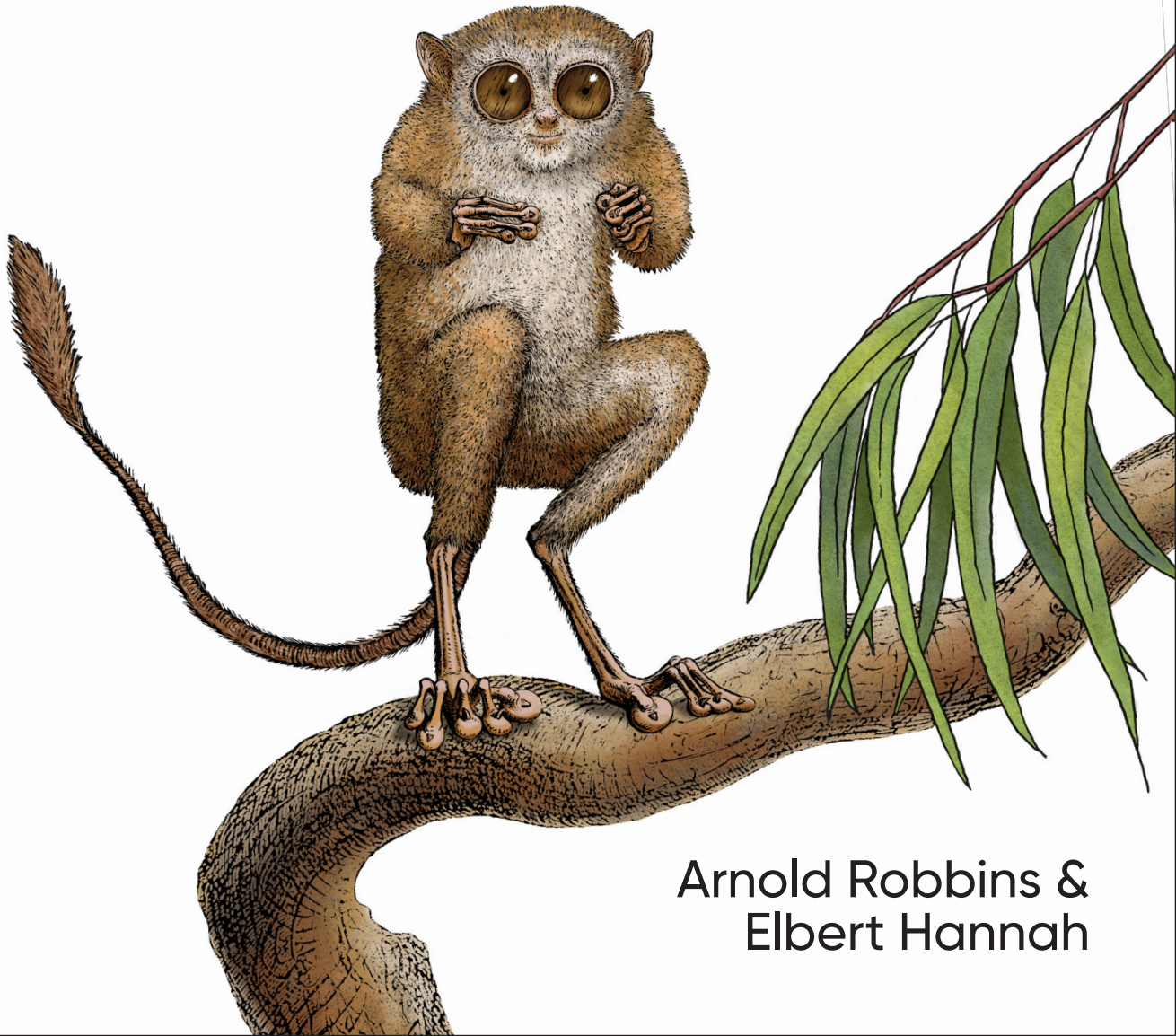8th Edition

Learning the vi & Vim Editors

Power and Agility Beyond Just Text Editing

Arnold Robbins &
Elbert Hannah

O'REILLY®

Learning the vi and Vim Editors

Among the text editors being used in the programming community, perhaps the most important family is vi and its derivatives. With this updated edition, Unix and Linux users will learn text editing basics for both vi and Vim ("vi improved") before moving on to advanced editing tools for each editor. Authors Arnold Robbins and Elbert Hannah cover the latest major releases of Vim, including 8.0 and 8.2.

If you're a programmer or computer analyst, or you work with browsers or command-line interfaces, using Vim can speed up your work and make complex tasks easier. You'll examine multiwindow editing, global search and replacement, and power tools for programmers, and learn how to write interactive macros and scripts to extend the editor—all in the easy-to-follow style that's made this book a classic.

- Go beyond the basics to learn which vi commands fit your specific needs
- Learn advanced vi tools that shift most of the editing burden to the computer
- Explore Vim tools that provide major improvements over vi
- Examine Vim's multiwindow editing feature, a significant upgrade over vi
- Use Vim scripts to customize and tailor Vim to your needs
- Look at Vim in modern GUI environments with Graphical Vim (gvim)
- See Vim in the broader programming milieu, including using it as an IDE

"In the midst of all the new tricks in this book, learning about Vim's outline and folding mode gave me the excuse I needed to finally stop using Emacs. That alone was worth it."

—Robert P. J. Day
Long-time Linux user and professional open source trainer since Slackware 1.00 came on 24 floppies

Arnold Robbins is a professional programmer and technical author. He's worked with Unix systems since 1980, including commercial Unix systems from Sun, IBM, HP, and DEC, and he's worked with GNU/Linux systems since 1996. He is also the long-time maintainer of GNU Awk (gawk) and its documentation as well as the author and coauthor of multiple O'Reilly titles.

Elbert Hannah specializes in integrating technologies. After discovering Unix early on, he built a career integrating disparate systems, ultimately integrating technologies in the finance industry. Underlying all of his accomplishments is a reliance on the power and agility of the vi text editing framework.

55999

9 781492 078807

Learning the vi and Vim Editors
Power and Agility Beyond Just Text Editing

Arnold Robbins and Elbert Hannah

Learning the vi and Vim Editors

by Arnold Robbins and Elbert Hannah

Printed in the United States of America.

To my wife, Miriam, for your love, patience, and support.

—Arnold Robbins, Sixth, Seventh, and Eighth Editions

To my wife, Anna, for your love, encouragement, and patience.
Thank you for being there.

—Elbert Hannah, Seventh and Eighth Editions

Table of Contents

Preface

Text editing is one of the most common tasks on any computer system, and vi is one of the most useful standard text editors on a system. With vi you can create new files or edit any existing text-only file.

vi, like many of the classic utilities developed during the early years of Unix®, has a reputation for being hard to navigate. Bram Moolenaar's enhanced clone, Vim ("vi Improved"), has gone a long way toward removing reasons for such impressions. Vim includes countless conveniences, visual guides, and help screens.

Today, Vim is the most popular version of vi, so this eighth edition focuses on Vim as follows:

- Part I, "vi and Vim Fundamentals", teaches basic vi skills, applicable to all versions of vi, but it does so in the context of Vim.
- Part II, "Vim", devotes a number of chapters specifically to Vim's advanced features.
- Part III, "Vim in the Larger Milieu", presents chapters relating to Vim in a larger context.

Scope of This Book

This book consists of 17 chapters and four appendixes, divided into four parts. Part I, "vi and Vim Fundamentals", is designed to get you started using vi and Vim quickly, and to follow up with advanced skills that will let you use them effectively.

The first two chapters—Chapter 1, "Introducing vi and Vim", and Chapter 2, "Simple Editing"—present some simple editing commands with which you can get started. You should practice these until they are second nature. You could stop reading at the end of Chapter 2, having learned some elementary editing operations.

But the editors are meant to do a lot more than rudimentary word processing; the variety of commands and options enables you to shortcut a lot of editing drudgery. Chapter 3, "Moving Around in a Hurry", and Chapter 4, "Beyond the Basics", concentrate on easier ways to do tasks. During your first reading, you'll get at least an idea of what `vi` and Vim can do and what commands you might harness for your specific needs. Later, you can come back to these chapters for further study.

Chapter 5, "Introducing the ex Editor", Chapter 6, "Global Replacement", and Chapter 7, "Advanced Editing", provide tools that help you shift more of the editing burden to the computer. They introduce you to the `ex` line editor underlying `vi` and Vim, and they show you how to issue `ex` commands from within `vi` and Vim.

Part II, "Vim", describes Vim, the most popular `vi` clone 21 years into the 21st century. It goes into detail on the many (many!) features Vim has over the original `vi`.

Chapter 8, "Vim (vi Improved): Overview and Improvements over vi", provides a general introduction to Vim. The chapter also gives an overview of the major improvements in Vim over `vi`, such as built-in help, control over initialization, additional motion commands, extended regular expressions, and many more.

Chapter 9, "Graphical Vim (gvim)", looks at Vim in modern GUI environments, such as those that are now standard on commercial Unix systems, GNU/Linux and other Unix work-alikes, and MS-Windows.

Chapter 10, "Multiple Windows in Vim", focuses on multiwindow editing, which is perhaps the most significant additional feature over standard `vi`. This chapter provides all the details on creating and using multiple windows.

Chapter 11, "Vim Enhancements for Programmers", focuses on Vim's use as a programmer's editor, above and beyond its facilities for general text editing. Of particular value are the folding and outlining facilities, smart indenting, syntax highlighting, and edit-compile-debug cycle speedups.

Chapter 12, "Vim Scripts", looks into the Vim command language, which lets you write scripts to customize and tailor Vim to suit your needs. Much of Vim's ease of use "out of the box" comes from the large number of scripts that other users have already written and contributed to the Vim distribution.

Chapter 13, "Other Cool Stuff in Vim", is a bit of a catchall chapter, covering a number of interesting points that don't fit into the earlier chapters.

Chapter 14, "Some Vim Power Techniques", presents some useful "power techniques." Based around the idea of personal key remappings, it shows you more ways to be productive.

Part III, "Vim in the Larger Milieu", looks at `vi`'s and Vim's roles in the larger software development and computer usage worlds.

Chapter 15, "Vim as IDE: Some Assembly Required", touches the tip of the iceberg of the world of Vim plug-ins, focusing on how you can change Vim from "just" an editor into a full-fledged integrated development environment (IDE).

Chapter 16, "vi Is Everywhere", looks at other significant software environments where vi-style editing can be brought into play to increase productivity.

Chapter 17, "Epilogue", provides a brief summary to round things off.

Part IV, "Appendixes", provides useful reference material.

Appendix A, "The vi, ex, and Vim Editors", lists all standard vi and ex commands, sorted by function. It also provides an alphabetical list of ex commands. Selected vi and ex commands from Vim are also included.

Appendix B, "Setting Options", lists set command options for vi and for Vim.

Appendix C, "The Lighter Side of vi", presents some humorous material related to vi.

Appendix D, "vi and Vim: Source Code and Building", describes where to get the "Heirloom" vi, as well as how to get Vim for your Unix, GNU/Linux, MS-Windows, or Macintosh system.

How the Material Is Presented

Our philosophy is to give you a good overview of what we feel are vi and Vim survival materials for the new user. Learning a new editor, especially an editor with all the options of Vim, can seem like an overwhelming task. We have made an effort to present basic concepts and commands in an easy-to-read and logical manner.

After providing the basics for vi and Vim, which are usable everywhere, we move on to cover Vim in depth. The following sections describe the conventions used in this book.

Discussion of vi Commands

For each keyboard command or group of related commands, you will find a brief introduction to the main concept before it is broken down into task-oriented sections. We then present the appropriate command to use in each case, along with a description of the command and the proper syntax for using it.

Conventions

In syntax descriptions and examples, what you would actually type is shown in the constant width font, as are all command names and program options. Variables (which you would not type literally but would replace with an actual value when you

type the command) are shown in `Constant width italic`. Brackets indicate that a variable is optional. For example, in the syntax line:

```
vi [filename]
```

`filename` would be replaced by an actual filename. The brackets indicate that the `vi` command can be invoked without specifying a filename at all. The brackets themselves are not typed.

Certain examples show the effect of commands typed at the shell prompt. In such examples, what you actually type is shown in **constant width bold**, to distinguish it from the system response. For example:

```
$ ls
ch01.xml ch02.xml ch03.xml ch04.xml
```

In code examples, *italic* indicates a comment that is not to be typed. In the text, *italic* refers to filenames, introduces special terms, and emphasizes anything that needs emphasis.

Following traditional Unix documentation convention, references of the form *printf*(3) refer to the online manual (accessed via the `man` command). This example refers to the entry for the `printf()` function in section 3 of the manual. You would type `man -s 3 printf` on most systems to see it.

Keystrokes

Special keystrokes are shown in a box. For example:

```
iWith a ESC
```

Throughout the book, you will also find columns of `vi`/Vim commands and their results:

Keystrokes	Results
ZZ	`"practice" [New] 6L, 104C written`
	Give the write and save command, ZZ. Your file is saved as a regular disk file.

In the preceding example, the command ZZ is shown in the left column. In the column to the right is a line (or several lines) of the screen that shows the result of the command. Cursor position is shown in reverse video. In this instance, since ZZ saves and writes the file, you see the status line shown when a file is written; the cursor position is not shown. Below the command/result is an explanation of the command and what it does.

In some of these demos, we show shell commands and their results. In such cases, the commands are preceded by the standard `$` shell prompt and the command is in bold:

Keystrokes	Results
`$ ls`	`ch01.asciidoc` `ch02.asciidoc` `ch03.asciidoc`

Sometimes `vi` commands are issued by pressing the CTRL key and another key simultaneously. In the text, this combination keystroke is usually written within a box (for example, CTRL-G). In code examples, it is written by preceding the name of the key with a caret (`^`). For example, `^G` means to hold down CTRL while pressing the G key. It is universal convention to refer to control characters using uppercase letters (`^G`, not `^g`) even though you do *not* hold down the SHIFT key when typing them.[1]

Additionally, when uppercase letters are shown using the keycap notation, we do so as SHIFT-X for any character *X*. Thus, `a` is represented as A, and A is represented as SHIFT-A.

Cautions, Notes, and Tips

This is a cautionary note. It describes things you need to watch out for or be careful about.

This is just a plain, regular old note. It point outs things that may be of interest or that may not have been obvious.

This is a tip. It provides helpful shortcuts or time-saving things you can do.

Problem Checklists

A problem checklist is included in those sections where you may run into some trouble. You can skim these checklists and go back to them when you actually encounter a problem.

---

[1] Perhaps this is because keyboards have the uppercase letters on the keys, not the lowercase ones.

What You Need to Know Before Starting

This book assumes that you have basic Unix user-level knowledge. In particular, you should already know how to:

- Open a terminal window on your laptop or workstation to get to a shell prompt
- Log in and log out, typically via `ssh`, if using a remote system
- Enter shell commands
- Change directories
- List files in a directory
- Create, copy, and remove files

Familiarity with `grep` (a global search program) and wildcard characters is also helpful.

Although modern systems let you run Vim from a GUI menu system, you lose access to the flexibility provided by Vim's command-line options. Thus, throughout the book, our examples continue to demonstrate running `vi` and Vim from the command-line prompt.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://www.github.com/learning-vi/vi-files*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning the vi and Vim Editors* by Arnold Robbins and Elbert Hannah (O'Reilly). Copyright 2022 Elbert Hannah and Arnold Robbins, 978-1-492-07880-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/viVim8*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

About the Previous Editions

In the fifth edition of this book (then called *Learning the vi Editor*), the ex editor commands were first discussed more fully. In Chapters 5, 6, and 7, the complex features of ex and vi were clarified by adding more examples, covering topics such as regular expression syntax, global replacement, *.exrc* files, word abbreviations, keyboard maps, and editing scripts. A few of the examples were drawn from articles in *UnixWorld* magazine. Walter Zintz wrote a two-part tutorial[2] on vi that taught us a few things we didn't know, and that also had a lot of clever examples illustrating features we did already cover in the book. Ray Swartz also had a helpful tip in one of his columns.[3] We are grateful for the ideas in these articles.

The sixth edition of *Learning the vi Editor* introduced coverage of four freely available "clones," or work-alike editors. Many of them have improvements over the original vi. One could thus say that there is a "family" of vi editors, and the book's goal was to teach you what you needed to know to use them. That edition treated nvi, Vim, elvis, and vile equally. A new appendix described vi's place in the larger Unix and internet culture.

The seventh edition of *Learning the vi and Vim Editors* retained all the good features of the sixth edition. Time had proven Vim to be the most popular vi clone, so the seventh edition added considerably expanded coverage of that editor (and gave Vim a place in the title). However, to be relevant for as many users as possible, it retained and updated the material on nvi, elvis, and vile.

About the Eighth Edition

This eighth edition of *Learning the vi and Vim Editors* retains all the good features of the seventh edition. Vim now "rules the roost," so this edition updates the coverage of Vim and removes the material on nvi, elvis, and vile. Part I now uses Vim as the context for its instruction and examples. Furthermore, references to problems with older versions of the original vi that are simply no longer relevant have been removed. We have attempted to streamline the book and keep it as relevant and useful as possible.

What's New

The following features are new for this edition:

---

2 "vi Tips for Power Users," *UnixWorld*, April 1990; and "Using vi to Automate Complex Edits," *UnixWorld*, May 1990. Both articles are by Walter Zintz.

3 Ray Swartz, "Answers to Unix," *UnixWorld*, August 1990.

- Once again, we have corrected errors in the basic text.
- We have thoroughly revised and updated the material in Part I and Part II. In Part I, we shifted the emphasis from the original Unix version of vi to being "vi in the context of Vim." We also added a new chapter to Part II.
- The additional chapters in Part III are brand new.
- We have changed the focus of Appendix C.
- We have moved the material on getting or building Vim from the mainline text to Appendix D.
- The other appendixes have been updated as well.

Versions

The following programs were used for testing out various vi features:

- The "Heirloom" vi from *https://github.com/n-t-roff/heirloom-ex-vi* served as the reference version of the original Unix vi.
- Solaris 11 `/usr/xpg7/bin/vi`. (On Solaris 11, `/usr/bin/vi` is actually Vim! The versions of vi in `/usr/xpg4/bin`, `/usr/xpg6/bin`, and `/usr/xpg7/bin` appear to be derived from the original Unix vi.)
- Versions 8.0, 8.1, and 8.2 of Bram Moolenaar's Vim.

Acknowledgments from the Sixth Edition

First and foremost, thanks to my wife, Miriam, for taking care of the kids while I was working on this book, particularly during the "witching hours" right before meal times. I owe her large amounts of quiet time and ice cream.

Paul Manno, of the Georgia Tech College of Computing, provided invaluable help in pacifying my printing software. Len Muellner and Erik Ray of O'Reilly & Associates helped with the SGML software. Jerry Peek's vi macros for SGML were invaluable.

Although all of the programs were used during the preparation of the new and revised material, most of the editing was done with Vim versions 4.5 and 5.0 under GNU/Linux (Red Hat 4.2).

Thanks to Keith Bostic, Steve Kirkendall, Bram Moolenaar, Paul Fox, Tom Dickey, and Kevin Buettner, who reviewed the book. Steve Kirkendall, Bram Moolenaar, Paul Fox, Tom Dickey, and Kevin Buettner also provided important parts of Chapters 8 through 12. (These chapter numbers refer to the sixth edition.)

Without the electricity being generated by the power company, doing anything with a computer is impossible. But when the electricity is there, you don't stop to think

about it. So too when writing a book—without an editor, nothing happens, but when the editor is there doing her job, it's easy to forget about her. Gigi Estabrook at O'Reilly is a true gem. It's been a pleasure working with her, and I appreciate everything she's done and continues to do for me.

Finally, many thanks to the production team at O'Reilly & Associates.

> Arnold Robbins
> Ra'anana, ISRAEL
> June 1998

Acknowledgments from the Seventh Edition

Once again, Arnold thanks his wife, Miriam, for her love and support. The size of his quiet time and ice cream debt continues to grow. In addition, thanks to J. D. "Illiad" Frazer for the great *User Friendly* cartoons.[4]

Elbert would like to thank Anna, Cally, Bobby, and his parents for staying excited about his work through the tough times. Their enthusiasm was contagious and appreciated.

Thanks to Keith Bostic and Steve Kirkendall for providing input on revising their editors' chapters. Tom Dickey provided significant input for revising the chapter on `vile` and the table of `set` options in Appendix B. Bram Moolenaar (the author of Vim) reviewed the book this time around as well. Robert P. J. Day, Matt Frye, Judith Myerson, and Stephen Figgins provided important review comments throughout the text.

Arnold and Elbert would both like to thank Andy Oram and Isabel Kunkle for their work as editors, and all of the tools and production staff at O'Reilly Media.

> Arnold Robbins
> Nof Ayalon
> ISRAEL
> April 2008
>
> Elbert Hannah
> Kildeer, Illinois
> USA
> April 2008

---

4 See *http://www.userfriendly.org* if you've never heard of User Friendly.

Acknowledgments for the Eighth Edition

We would like to thank Krishnan Ravikumar, whose email to Arnold asking about a new edition started the ball rolling to update the book.

We would also like to thank our technical reviewers (in alphabetical order): Yehezkel Bernat, Robert P. J. Day, Will Gallego, Jess Males, Ofra Moyal-Cohen, Paul Pomerleau, and Miriam Robbins.

Arnold would like to thank his wife, Miriam, yet again, for her doing without him while the book was going on. He also thanks his children, Chana, Rivka, Nachum, and Malka, as well as Sophie the dog.

Elbert would like to thank the following:

- His wife, Anna, who *again* accepted the odd schedule and demands of putting this book together. He also thanks Bobby and Cally for their support and encouragement as the work progressed. Their always-cheerful attitude always uplifted. And he extends a special thank-you to new grandson Dean. One of Dean's first words was "book," and Elbert can only assume Dean was referring to this one.

- His West Highland Terrier, Poncho, who was there when he wrote the seventh edition and is still alive and kicking and eagerly awaiting the eighth edition. He doesn't know how to read but he still "gets" Vim. Good boy, Poncho! Paws always on the keyboard, never touching the mouse.

- His CME group peers for a great 13 years in which he honed his Vim skills and taught others the Vim greatness.

  — He gives special mention to Scott Fink, a peer, a boss, a collaborator, and a friend, who always asked to learn more not only about Vim but about all things in the Vim universe. Working with Scott, he harnessed Vim "zen" to write great applications together.

  — Paul Pomerleau for being a technical reviewer of this book *and* someone who always kept him honest about the Vim/Emacs comparison. And even though Paul used Emacs, he was one of Elbert's greatest collaborators and friends those 13 years.

  — Michael Sciacco for showing him Microsoft's VS Code. Michael taught this old dog a lot of new tricks. Michael, *you're* an IDE!

  — Finally, Tony Ferraro, under whom he worked his last professional days. Tony always encouraged Elbert to write (technical documentation), and Elbert tried. This book is for you, Tony!

Both of us would like to thank our editors for this edition, Gary O'Brien and Shira Evans, for patiently shepherding us through the revision process. Managing programmers has been said to be akin to herding cats; no doubt the same applies to managing authors. Similarly, we thank the tools and production staff at O'Reilly Media.

Arnold Robbins
Nof Ayalon
ISRAEL
September 2021

Elbert Hannah
Kildeer, Illinois
USA
September 2021

vi and Vim Fundamentals

Part I is designed to get you started quickly with the vi and Vim editors. It provides the advanced skills that will let you use them most effectively. These chapters cover the functionality of the original, core vi, providing commands you can use on any version. Later chapters cover advanced features in Vim. This part contains the following chapters:

- Chapter 1, "Introducing vi and Vim"
- Chapter 2, "Simple Editing"
- Chapter 3, "Moving Around in a Hurry"
- Chapter 4, "Beyond the Basics"
- Chapter 5, "Introducing the ex Editor"
- Chapter 6, "Global Replacement"
- Chapter 7, "Advanced Editing"

Introducing vi and Vim

One of the most important day-to-day uses of a computer is working with text: composing new text, editing and rearranging existing text, deleting or rewriting incorrect and obsolete text. If you work with a word processing program such as Microsoft Word, that's what you're doing! If you are a programmer, you're also working with text: the source code files of your program, and auxiliary files needed for development. Text editors process the contents of any text files, whether those files contain data, source code, or sentences.

This book is about text editing with two related text editors: `vi` and Vim. `vi` has a long tradition as the standard Unix[1] text editor. Vim builds on `vi`'s command mode and command language, providing at least an order of magnitude more power and capability than the original.

Text Editors and Text Editing

Let's get started.

Text Editors

Unix text editors have evolved over time. Initially, there were *line editors*, such as `ed` and `ex`, for use on serial terminals that printed on continuous feed paper. (Yes,

---

1 These days, the term "Unix" includes both commercial systems derived from the original Unix code base and Unix work-alikes whose source code is available. Solaris, AIX, and HP-UX are examples of the former, and GNU/Linux and the various BSD-derived systems are examples of the latter. Also included under this umbrella are macOS's terminal environment, Windows Subsystem for Linux (WSL) on MS-Windows, and Cygwin and other similar environments for Windows. Unless otherwise noted, everything in this book applies across the board to all those systems.

people really programmed on such things! Including at least one of your authors.) Line editors were called such because you worked on your program one or a few lines at a time.

With the introduction of cathode-ray tube (CRT) terminals with cursor addressing, line editors evolved into *screen editors*, such as `vi` and Emacs. Screen editors let you work with your files a full screen at a time and let you easily move around the lines on the screen as you wished.

With the introduction of graphical user interface (GUI) environments, screen editors evolved further into graphical text editors, where you use a mouse to scroll the visible portion of your file, move to a particular point in a file, and select text upon which to perform an operation. Examples of such text editors based on the X Window System are `gedit` on Gnome-based systems and Notepad++ on MS-Windows. There are others.

Of particular interest to us is that the popular screen editors have evolved into graphical editors:[2] GNU Emacs provides multiple X windows, as does Vim through its `gvim` version. The graphical editors continue to work identically to their original screen-based versions, making the transition to the GUI version almost trivial.

Of all the *standard* editors on a Unix system, `vi` is the most useful one for you to master.[3] Unlike Emacs, it is available in nearly identical form on every modern Unix system, thus providing a kind of text-editing lingua franca.[4] The same might be said of `ed` and `ex`, but screen editors, and their GUI-based descendants, are much easier to use. (So much so, in fact, that line editors have generally fallen into disuse.)

`vi` exists in multiple incarnations. There is the original Unix version, and there are multiple "clones": programs written from scratch to behave as `vi` does, but not based on the original `vi` source code. Of these, Vim has become the most popular.

In the chapters in Part I, we teach you how to use `vi` in the general sense. Everything in these chapters applies to all versions of `vi`. However, we do this in the context of Vim, since that is the version you are likely to have on your system. While reading, feel free to think of "`vi`" as standing for "`vi` and Vim."

---

2 Perhaps in the same way as Pokémon do?

3 If you don't have either `vi` or Vim installed, see Appendix D, "vi and Vim: Source Code and Building".

4 GNU Emacs has become the universal version of Emacs. The only problem is that it doesn't come standard with most systems; you must retrieve and install it yourself, even on some GNU/Linux systems.

`vi` is short for *visual* editor and is pronounced "vee-eye." This is illustrated graphically in Figure 1-1.
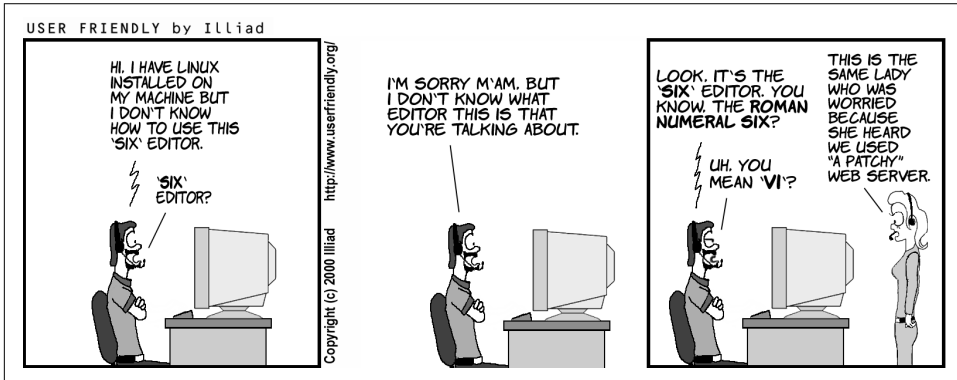
Figure 1-1. Correct pronunciation of `vi`

To many beginners, `vi` looks unintuitive and cumbersome—instead of using special control keys for word processing functions and just letting you type normally, it uses almost all of the regular keyboard keys for issuing commands. When the keyboard keys are issuing commands, the editor is said to be in *command mode*. You must be in a special *insert mode* before you can type actual text on the screen. In addition, there seem to be so many commands.

Once you start learning, however, you realize that the editor is well designed. You need only a few keystrokes to tell it to do complex tasks. As you learn `vi`, you learn shortcuts that transfer more and more of the editing work to the computer—where it belongs.

`vi` and Vim (like any text editors) are not "what you see is what you get" word processors. If you want to produce formatted documents, you must type in specific instructions (sometimes called *formatting codes*) that are used by a separate formatting program to control the appearance of the printed copy. If you want to indent several paragraphs, for instance, you put a code where the indent begins and ends. Formatting codes allow you to experiment with or change the appearance of your printed files, and in many ways they give you much more control over the appearance of your documents than a word processor does.

Formatting codes are the specific verbs in what are more generally known as *markup languages*.[5] In recent years, markup languages have seen a resurgence in popularity. Of note are Markdown and AsciiDoc,[6] although there are others as well. Perhaps the most widely used markup language today is the Hypertext Markup Language (HTML), used in the creation of internet web pages.

Besides the markup languages just mentioned, Unix supports the `troff` formatting package.[7] The TeX and LaTeX formatters are popular, commonly available alternatives. The easiest way to use any of these markup languages is with a text editor.

> `vi` does support some simple formatting mechanisms. For example, you can tell it to automatically wrap when you come to the end of a line, or to automatically indent new lines. In addition, Vim provides automatic spellchecking.

As with any skill, the more editing you do, the easier the basics become, and the more you can accomplish. Once you are used to all the powers you have while editing, you may never want to return to any "simpler" editor.

Text Editing

What are the components of editing? First, you want to *insert* text (a forgotten word or a new or missing sentence), and you want to *delete* text (a stray character or an entire paragraph). You also need to *change* letters and words (to correct misspellings or to reflect a change of mind about a term). You might want to *move* text from one place to another part of your file. And, on occasion, you want to *copy* text to duplicate it in another part of your file.

Unlike many word processors, `vi`'s command mode is the initial or default mode. Complex, interactive edits can be performed with only a few keystrokes. To insert raw text, you simply give any of the several "insert" commands and then type away.

---

5 From the use of red pencils to "mark up" changes in typeset galleys or proofs.

6 For more information on these languages, see *https://en.wikipedia.org/wiki/Markdown* and *http://asciidoc.org*, respectively. This book is written in AsciiDoc.

7 `troff` is for laser printers and typesetters. Its "twin brother" is `nroff`, for line printers and terminals. Both accept the same input language. Following common Unix convention, we refer to both with the name `troff`. Today, anyone using `troff` uses the GNU version, `groff`.

One or two characters are used for the basic commands. For example:

`i`

Insert

`cw`

Change word

Using letters as commands, you can edit a file with great speed. You don't have to memorize banks of function keys or stretch your fingers to reach awkward combinations of keys. You never have to remove your hands from the keyboard, or mess around with multiple levels of menus! Most of the commands can be remembered by the letters that perform them. Nearly all commands follow similar patterns and are related to each other.

In general, `vi` and Vim commands:

- Are case sensitive (uppercase and lowercase keystrokes mean different things; `I` is different from `i`).
- Are not shown (or "echoed") on the screen when you type them.
- Do not require that you press ENTER after a command.

There is also a group of commands that echo on the bottom line of the screen. Bottom-line commands are preceded by different symbols. The slash (`/`) and the question mark (`?`) begin search commands and are discussed in Chapter 3, "Moving Around in a Hurry". A colon (`:`) begins all `ex` commands. `ex` commands are those used by the `ex` line editor. The `ex` editor is available to you when you use any version of `vi`, because `ex` is the underlying editor and `vi` is really just its "visual" mode. `ex` commands and concepts are discussed fully in Chapter 5, "Introducing the ex Editor", but this chapter introduces you to the `ex` commands to quit a file without saving edits.

A Brief Historical Perspective

Before we dive into all the ins and outs of `vi` and Vim, it will help to understand `vi`'s worldview of your environment. In particular, this will help you make sense of many of `vi`'s otherwise more obscure error messages and also appreciate how Vim has evolved beyond the original `vi`.

`vi` dates back to a time when computer users worked on CRT terminals connected via serial lines to central minicomputers. Hundreds of different kinds of terminals existed and were in use worldwide. Each one did the same kind of actions (clear the screen, move the cursor, etc.), but the commands needed to make them do these actions were different. In addition, the Unix system let you choose the characters

to use for backspace, generating an interrupt signal, and other commands useful on serial terminals, such as suspending and resuming output. These facilities were (and still are) managed with the `stty` command.

The original Berkeley Unix version of `vi` abstracted out the terminal control information from the code (which was hard to change) into a text-file database of **term**inal **cap**abilities (which was easy to change), managed by the `termcap` library. In the early 1980s, System V introduced a binary **term**inal **info**rmation database and `terminfo` library. The two libraries were largely functionally equivalent. In order to tell `vi` which terminal you had, you had to set the `TERM` environment variable. This was typically done in a shell startup file such as your personal *.profile* or *.login*.

The `termcap` library is no longer used. GNU/Linux and BSD systems use the `ncurses` library, which provides a compatible superset of the System V `terminfo` library's database and capabilities.

Today, everyone uses terminal emulators in a graphical environment (such as Gnome Terminal). The system almost always takes care of setting `TERM` for you.

> You can use Vim from a PC non-GUI console too, of course. This is very useful when doing system recovery work in single-user mode. There aren't too many people left who would want to work this way on a regular basis, though.

For day-to-day use, it is likely that you will want to use a GUI version of `vi`, such as `gvim`. On a Microsoft Windows or macOS system, this will probably be the default. However, when you run `vi` (or some other screen editor of the same vintage) inside a terminal emulator, it still uses `TERM` and `terminfo` and pays attention to the `stty` settings. And using it inside a terminal emulator is just as easy a way to learn `vi` and Vim as any other.

Another important fact to understand about `vi` is that it was developed at a time when Unix systems were considerably less stable than they are today. The `vi` user of yesteryear had to be prepared for the system to crash at arbitrary times, and so `vi` included support for recovering files that were in the middle of being edited when the system crashed.[8] So, as you learn `vi` and Vim and see the descriptions of various problems that might occur, bear these historical developments in mind.

---

8 Thankfully, this kind of thing is much less common, although systems can still crash due to external circumstances, such as a power outage. If you have an uninterruptible power supply for your system, or a healthy battery on your laptop, even this worry goes away.

Opening and Closing Files

You can use `vi` to edit any text file. The editor copies the file to be edited into a *buffer* (an area temporarily set aside in memory), displays the buffer (though you can see only one screenful at a time), and lets you add, delete, and change text. When you save your edits, the editor copies the edited buffer back into a permanent file, replacing the old file of the same name. Remember that you are always working on a *copy* of your file in the buffer, and that your edits do not affect your original file until you save the buffer. Saving your edits is also called "writing the buffer," or more commonly, "writing your file."

Opening a File from the Command Line

`vim` is the Unix command that invokes the Vim editor for an existing file or for a brand-new file. The syntax for the `vim` command is:

    $ **vim** [*filename*]

or

    $ **vi** [*filename*]

On modern systems, `vi` is often just a link to Vim. The brackets shown on these command lines indicate that the filename is optional. The brackets should not be typed. The `$` is the shell prompt.

If the filename is omitted, the editor opens an unnamed buffer. You can assign the name when you write the buffer into a file. For right now, though, let's stick to naming the file on the command line.

A filename must be unique inside its directory. (Some operating systems call directories *folders*; they're the same thing.)

On Unix systems, a filename can include any 8-bit character except a slash (/), which is reserved as the separator between files and directories in a pathname, and ASCII NUL, the character with all zero bits. You can even include spaces in a filename by typing a backslash (\) before the space. (MS-Windows systems disallow the backslash [\] and the colon [:] character in filenames.) In practice, though, filenames generally consist of any combination of uppercase and lowercase letters, numbers, and the characters dot (.) and underscore (_). Remember that Unix is case sensitive: lowercase letters are distinct from uppercase letters. Also remember that you must press ENTER to tell the shell that you are finished issuing your command.

When you want to open a new file in a directory, give a new filename with the `vi` command. For example, if you want to open a new file called *practice* in the current directory, you would enter:

    $ **vi practice**

Since this is a new file, the buffer is empty, and the screen appears as follows:

```
~
~
~
"practice" [New file]
```

The tildes (~) down the lefthand column of the screen indicate that there is no text in the file, not even blank lines. The prompt line (also called the status line) at the bottom of the screen echoes the name and status of the file.

You can also edit any existing text file in a directory by specifying its filename. Suppose that there is a Unix file with the pathname */home/john/letter*. If you are already in the */home/john* directory, use the relative pathname. For example:

```
$ vi letter
```

brings a copy of the file *letter* to the screen.

If you are in another directory, give the full pathname to begin editing:

```
$ vi /home/john/letter
```

Opening a File from the GUI

Although we (strongly) recommend that you become comfortable with the command line, you can run Vim on a file directly from your GUI environment. Typically, you right-click on a file and then select something like "Open with …" from the menu that pops up. If Vim is correctly installed, it will be one of the available options for opening the file.

Usually, you may also start Vim directly from your menuing system, in which case you then need to tell it which file to edit with the ex command `:e` `filename`.

We can't be any more specific than this, because there are so many different GUI environments in use today.

Problems Opening Files

- *You see one of the following messages:*

  ```
  Visual needs addressable cursor or upline capability
  terminal: Unknown terminal type
  Block device required
  Not a typewriter
  ```

  Your terminal type is undefined, or else there's probably something wrong with your `terminfo` entry. Enter `:q` to quit. Often, setting `$TERM` to `vt100` is enough to get going, at least in a bare-bones sort of fashion. For further help, you might use an internet search engine or a popular technical questions forum such as Stack Overflow.

- *A* `[new file]` *message appears when you think a file already exists.*

  Check that you have used correct case in the filename (Unix filenames are case sensitive). If you have, then you are probably in the wrong directory. Enter `:q` to quit. Then check to see that you are in the correct directory for that file (enter `pwd` at the shell prompt). If you are in the right directory, check the list of files in the directory (with `ls`) to see whether the file exists under a slightly different name.

- *You invoke* `vi` *but you get a colon prompt (indicating that you're in* `ex` *line-editing mode).*

  You probably typed an interrupt (typically CTRL-C) before `vi` could draw the screen. Enter `vi` by typing `vi` at the `ex` prompt (`:`).

- *One of the following messages appears:*

      [Read only]
      File is read only
      Permission denied

  "Read only" means that you can only look at the file; you cannot save any changes you make. You may have invoked `vi` in `view` mode (with `view` or `vi -R`), or you do not have write permission for the file. See the section "Opening a File from the Command Line" on page 9.

- *One of the following messages appears:*

      Bad file number
      Block special file
      Character special file
      Directory
      Executable
      Non-ascii file
      file non-ASCII

  The file you've called up to edit is not a regular text file. Type `:q!` to quit, and then check the file you wish to edit, perhaps with the `file` command.

- *When you type* `:q` *because of one of the previously mentioned difficulties, this message appears:*

      E37: No write since last change (add ! to override)

  You have modified the file without realizing it. Type `:q!` to leave the editor. Your changes from this session are not saved in the file.

Modus Operandi

As mentioned earlier, the concept of the current "mode" is fundamental to the way `vi` works. There are two modes, *command mode* and *insert mode*. (The `ex` command mode can be considered a third mode, but we'll ignore that for now.) You start out

in command mode, where every keystroke represents a command.[9] In insert mode, everything you type becomes text in your file.

Sometimes, you can accidentally enter insert mode, or conversely, you might leave insert mode accidentally. In either case, what you type will likely affect your files in ways you did not intend.

Press the ESC key to force the editor to enter command mode. If you are already in command mode, the editor beeps at you when you press the ESC key. (Command mode is thus sometimes referred to as "beep mode.")

Once you are safely in command mode, you can proceed to repair any accidental changes and then continue editing your text. (See the section "Problems with deletions" on page 31, and also see the section "Undo" on page 35.)

Saving and Quitting a File

You can quit working on a file at any time, save your edits, and return to the command prompt (if you're running inside a terminal window). The command to quit and save edits is ZZ. Note that ZZ is capitalized.

Let's assume that you do create a file called *practice* to practice vi commands, and that you type in six lines of text. To save the file, first check that you are in command mode by pressing ESC, and then enter ZZ:

Keystrokes	Results
ZZ	`"practice" [New] 6L, 104C written`
	Give the write and save command, ZZ. Your file is saved as a regular disk file.
`$ ls`	`ch01.asciidoc      ch02.asciidoc      practice`
	Listing the files in the directory shows the new file *practice* that you created.

You can also save your edits with ex commands. Type :w to save (write) your file but not quit; type :q to quit if you haven't made any edits; and type :wq to both save your edits and quit. (:wq is equivalent to ZZ.) We'll explain fully how to use ex commands in Chapter 5; for now, you should just memorize a few commands for writing and saving files.

---

9 Note that vi and Vim do not have commands for every possible key. Rather, in command mode, the editor expects to receive keys representing commands, not text to go into your file. We take advantage of unused keys later, in the section "Using the map Command" on page 124.

Quitting Without Saving Edits

When you are first learning Vim, especially if you are an intrepid experimenter, there are two other `ex` commands that are handy for getting out of any mess that you might create.

What if you want to wipe out all the edits you have made in a session and then reload the original file? The command:

    `:e!` `ENTER`

reloads the last saved version of the file so you can start over.

Suppose, however, that you want to wipe out your edits and then just quit the editor? The command:

    `:q!` `ENTER`

immediately quits the file you're editing and returns you to the command prompt. With both of these commands, you lose all edits made in the buffer since the last time you saved the file. The editor normally won't let you throw away your edits. The exclamation point added to the `:e` or `:q` command causes it to override this prohibition, performing the operation even though the buffer has been modified.

Going forward, we don't show the `ENTER` key on `ex` mode commands, but you must use it to get the editor to execute them.

Problems Saving Files

- *You try to write your file, but you get one of the following messages:*

  ```
  File exists
  File file exists - use w!
  [Existing file]
  File is read only
  ```

  Type `:w!` *file* to overwrite the existing file, or type `:w` *newfile* to save the edited version in a new file.

- *You want to write a file, but you don't have write permission for it. You get the message "Permission denied."*

  Use `:w` *newfile* to write out the buffer into a new file. If you have write permission for the directory, you can use the `mv` command to replace the original version with your copy of it. If you don't have write permission for the directory, type `:w` *pathname/file* to write out the buffer to a directory for which you do have write permission (such as your home directory, or */tmp*). Be careful not to overwrite any existing files in that directory.

- *You try to write your file, but you get a message telling you that the file system is full.*

Today, when a 500-gigabyte drive is considered small, errors like this are generally rare. If something like this does occur, you have several courses you can take. First, try to write your file somewhere safe on a different file system (such as */tmp*) so that your data is saved. Then try to force the system to save your buffer with the ex command `:pre` (short for `:preserve`). If that doesn't work, look for some files to remove, as follows:

— Open a graphical file manager (such as Nautilus on GNU/Linux) and try to find old files you don't need and can remove.

— Use CTRL-Z to suspend `vi` and return to the shell prompt. You can then use various Unix commands to try to find large files that are candidates for removal:

  — `df` indicates how much **d**isk space is **f**ree on a given filesystem, or on the system as a whole.

  — `du` indicates how many **d**isk blocks are **u**sed for given files and directories. `du -s * | sort -nr` is an easy way to get a list of files and directories sorted by how much space they use in descending order.

When done removing files, use `fg` to put `vi` back into the foreground; you can then save your work normally.

While we're at it, besides using CTRL-Z and job control, you should know that you can type `:sh` to start a new shell in which to work. Type CTRL-D or `exit` to terminate the shell and return to `vi`. (This even works from with `gvim`!)

You can also use something like `:!du -s *` to run a shell command from within `vi` and then return to your editing when the command is done.

Exercises

The only way to learn `vi` and Vim is to practice. You now know enough to create a new file and to return to the command prompt. Create a file called *practice*, insert some text, and then save and quit the file.

Open a file called *practice* in the current directory:	`$ vi practice`
Enter insert mode:	`i`
Insert text:	`any text you like`
Return to command mode:	ESC
Quit `vi`, saving edits:	`ZZ`

Simple Editing

This chapter introduces you to editing with `vi` and Vim, and it is set up to be read as a tutorial. In it you will learn how to move the cursor and how to make some simple edits. If you've never worked with these editors, you should read the entire chapter.

Later chapters show you how to expand your skills to perform faster and more powerful edits. One of the biggest advantages for an adept user is that there are so many options to choose from. Of course, as with many advanced tools, one of the biggest *disadvantages* for the newcomer to `vi` and Vim is that there are so many different editor commands to learn.

You can't learn to use the editor by memorizing every single `vi` command. Start out by learning the basic commands introduced in this chapter. Note the patterns of use that the commands have in common. We point out these patterns as we encounter them.

As you learn, be on the lookout for more tasks that you can delegate to the editor, and then find the command that accomplishes it. In later chapters you will learn more advanced features of `vi` and Vim, but before you can handle the advanced, you must master the simple.

This chapter covers:

- Moving the cursor
- Simple edits: Adding, changing, deleting, moving, and copying text
- More ways to enter insert mode
- Joining lines
- Mode indicators

vi Commands

As we've seen, `vi` and Vim have two primary modes: *command mode* and *insert mode*. The command line (or colon prompt), where you issue `ex` commands, can be considered a third mode; its use is more advanced and is covered in later chapters.

When you first open a file, you are in command mode, and the editor is waiting for you to enter a command. Commands enable you to move anywhere in the file, to perform edits, or to enter insert mode to add new text. Commands can also be given to exit the file (saving or ignoring your edits) in order to return to the shell prompt.

You can think of the different modes as representing two different keyboards. In insert mode, your keyboard functions regularly. In command mode, each key has a new meaning or initiates some instruction.

There are several ways to tell Vim that you want to begin insert mode. One of the most common is to press `i`. The `i` doesn't appear on the screen, but after you press it, whatever you type *does* appear on the screen and is entered into the buffer. The cursor marks the current insertion point.[1] To tell Vim that you want to stop inserting text, press ESC. Pressing ESC moves the cursor back one space (so that it is on the last character you typed) and returns you to command mode.

For example, suppose you have opened a new file and want to insert the word "introduction." If you type the keystrokes `iintroduction`, what appears on the screen is:

```
introduction
```

When you open a new file, Vim starts in command mode and interprets the first keystroke (`i`) as the insert command. All keystrokes made after the insert command are considered text until you press ESC. If you need to correct a mistake while in insert mode, backspace and type over the error. Depending on your terminal and its settings, backspacing may erase what you've previously typed or may just back up over it. In either case, whatever you back up over is deleted. Note that you can't use the backspace key to back up beyond the point where you entered insert mode. (If you have disabled `vi` compatibility, Vim allows you to backspace beyond the point where you entered insert mode. Most GNU/Linux distributions have Vim set up with `vi` compatibility disabled, so this may work for you out of the box.)

Vim has an option that lets you define a right margin and provides a carriage return automatically when you reach it. For right now, while you are inserting text, press ENTER to break the lines.

---

1 Some versions show that you're in input mode in the status line. This is discussed in the section "Mode Indicators" on page 38.

Sometimes you don't know whether you are in insert mode or command mode. Whenever Vim does not respond as you expect, press ESC once or twice to check which mode you are in. When you hear the beep, you are in command mode.[2]

Moving the Cursor in Command Mode

You may spend only a small amount of time in an editing session adding new text in insert mode; much of the time you will be making edits to existing text by moving around your file and issuing commands.

In command mode you can position the cursor anywhere in the file. Since you begin all basic edits (changing, deleting, and copying text) by placing the cursor at the text that you want to change, you want to be able to move the cursor to that place as quickly as possible.

There are vi commands to move the cursor:

- Up, down, left, or right—one *character* at a time
- Forward or backward by blocks of *text* such as words, sentences, or paragraphs
- Forward or backward through a file, one *screen* at a time

In Figure 2-1, the s in reverse video in *seeing* on the third line marks the present cursor position. Circles show movement of the cursor from its current position to the position that would result from various vi commands.


```
                    2k
                     |
     With a screen editor you can scroll the
     page, move the cursor, delete lines,
  0——and more, while seeing the results of
     your edits as you make them.|
                |    |   |            |            |
                b    2h  j            2w           $
```
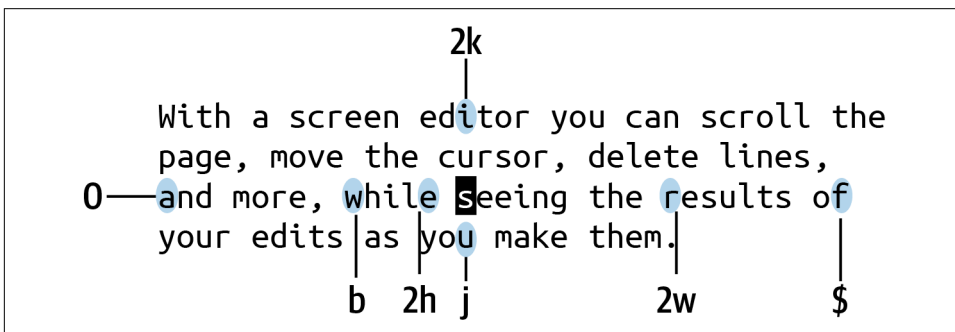
Figure 2-1. Sample movement commands starting from the central s

---

2 If you've muted the sound on your system, you'll have to work harder to determine what mode you're in by watching how the editor responds to what you type.

Single Movements

The keys h, j, k, and l, right under your fingertips, move the cursor as follows:

h

> Left one space

j

> Down one line

k

> Up one line

l

> Right one space

You can also use the cursor arrow keys (←, ↓, ↑, →), + and - to go up and down, CTRL-P and CTRL-N to also go up and down, or the ENTER and BACKSPACE keys, but they are all out of the way.

At first, it may seem awkward to use letter keys instead of arrows for cursor movement. After a short while, though, you'll find it is one of the things you'll like best about vi and Vim—you can move around without ever taking your fingers off the center of the keyboard.

Before you move the cursor, press ESC to make sure that you are in command mode. Use h, j, k, and l to move forward or backward in the file from the current cursor position. When you have gone as far as possible in one direction, you hear a beep and the cursor stops. For example, once you're at the beginning or end of a line, you cannot use h or l to wrap around to the previous or next line; you have to use j or k.[3] Similarly, you cannot move the cursor past a tilde (~) representing a line without text, nor can you move the cursor above the first line of text.

Why h, j, k, and l?

Mary Ann Horton, who has been involved with Berkeley Unix since almost the beginning, tells the following story:

While the vi experience was much like Notepad, it was also a very powerful editor. Students and faculty made heavy use of the power tools available, like the "global" command that would make the same change on all lines matching some pattern, or the ability to give commands like "delete 13 paragraphs" or "copy the text through the

---

3 Vim, with nocompatible set, allows you to "space past" the end of the line to the next one with l or the space bar. This is likely to be the default.

---

matching parenthesis." But vi had a steep learning curve, and first-time users wanted to use the arrow keys on their terminals to move around in the file, Notepad style.

Arrow keys didn't work in vi, and for a very good reason. Users had a variety of different brands of terminals, and all those terminals' arrow keys sent different codes when they were pressed.

Bill [Joy] didn't have to worry about arrow keys. He had found a way to work from home, getting a Lear-Siegler ADM-3A terminal in his apartment. The ADM-3A was widely advertised as "the dumb terminal" because it didn't have a lot of fancy features, like arrow keys, allowing it to be sold for the then-low price of $995. Instead, LSI painted arrows on the H, J, K, and L keys.[4] Bill had set up vi commands to match: h moved the cursor left, j down, k up, and l to the right. Every vi user had to learn h, j, k, and l to move around the file.

What if you wanted to type a word with an "h" in it? vi, like ed, was a "moded" editor. This meant you were either in "command mode," where it treated keys you pressed as commands, or "input mode," where keystrokes were content to be added to the file. A command like i for "insert" put you in input mode, and the Escape (ESC) key got you back to command mode.

How to get arrow keys to work in vi? These special keys sent two or three character sequences, usually beginning with Escape. We called them "escape sequences." Escape, however, was already an important vi command. It took you out of input mode, and if you were already out of input mode, it beeped. One of the first things you learned in vi was that, if you'd forgotten which mode you were in, you pressed the ESC key until it beeped, and then you knew you were in command mode.

vi used a terminal capability database file called "termcap," which told it which codes, for your specific model terminal, to send to move the cursor, clear the screen, and the like. It was easy enough to add the arrow key sequences to termcap.

If the computer received an Escape, was the user hitting the ESC key or an arrow key? Should the editor exit input mode, or should it wait for more text to interpret an arrow key? Once the editor tried to read more text, the program would hang until something came in.

Fortunately, a new Unix feature allowed the editor to wait briefly to see if another character came in. If that character might be part of a valid escape sequence, vi could keep reading to see what other key the user had pressed. If no more characters came in for that brief interval, the user must have pressed the ESC key. Problem solved!

Around the spring of 1979, I added code and termcap entries for vi to understand arrow keys, Home, Page-Up, and other keys that some of the terminals had. I

---

4 Pictures of this terminal's keyboard can be found easily by searching on the internet. —ADR

configured termcap as if the ADM-3a had arrow keys that sent h, j, k, and l; and then I deleted the hardcoded h, j, k, and l commands. I thought I had it all fixed up.

Within a day, I had a line of angry CS grad students outside my office door. Peter was at the head of the line. He wanted to know why I broke hjkl on his terminal. I explained to him that his arrow keys worked now, and he didn't have to use hjkl; he could use the arrow keys instead.

Peter rolled his eyes. "You don't understand," he said. "We like using hjkl! We're touch typists. Our fingers are right over the hjkl keys. We don't want to have to move them way to the edge of the keyboard to use arrow keys. Give us back our hjkl commands!" The line of students agreed.

They were right. I put back hjkl and left the arrow key functionality in too. And I realized how important the key placements of vi commands were. Almost any command you used often was a lowercase letter. I got really fast with vi, and to this day I prefer vi to edit text files. I've trained several classes of IT professionals how to get the most out of vi and Unix power tools.

Numeric Arguments

Often, you may wish to repeat a command multiple times. Instead of just typing the command over and over again, you can precede a command with numbers. This is known as a *repeat count* or *replication factor*.

Figure 2-2 shows how the command 4l moves the cursor four character positions to the right, just as if you had typed l four times (llll).
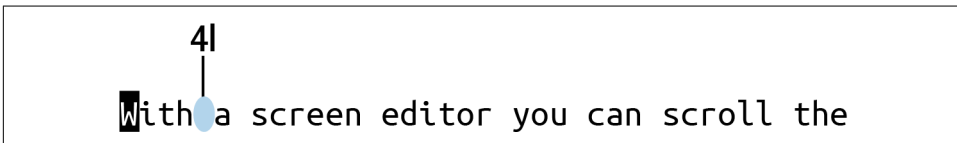
Figure 2-2. Multiplying commands by numbers

The repeat count comes *before* the command it quantifies because if it came after, vi would never know when the number was finished.

The ability to multiply commands gives you more options and power for each command you learn. Keep this in mind as we introduce additional commands.

Movement Within a Line

When you saved the file *practice*, Vim displayed a message telling you how many lines are in that file. A *line* is not necessarily the same length as the visible line that appears on the screen. A line is any text entered between newlines. (A *newline* character is

inserted into the file when you press the ENTER key in insert mode.) If you type 200 characters before pressing ENTER, Vim regards all of those characters as a single line (even though those characters visibly take up several lines on the screen).

We mentioned as an aside in Chapter 1, "Introducing vi and Vim", that vi and Vim have an option that allows you to set a distance from the right margin (the end of the line) at which they automatically insert a newline character. This option is wrapmargin (its abbreviation is wm). You can set a wrapmargin at 10 characters:

```
:set wm=10
```

This command doesn't affect lines that you've already typed. Once you've set this, try entering some new lines, and you'll see Vim automatically wrapping those lines for you, breaking the lines between words. We'll talk more about setting options in Chapter 7, "Advanced Editing", but this one really couldn't wait!

> If you put this command into a file named *.exrc* in your home directory, the editor will automatically execute it every time it starts up. We cover vi and Vim startup files later in the book.

If you do not use the automatic wrapmargin option, you should break lines with ENTER to keep the lines of manageable length.

Two useful commands that involve movement within a line are:

0 *(digit zero)*
    Move to beginning of line.

$
    Move to end of line.

Line numbers are displayed in the following example. (Line numbers can be displayed by using the number option, which is enabled by typing :set nu in command mode. This operation is described in Chapter 5.) Note that the line numbers are not part of the file's contents; the editor displays them for your convenience:

```
1  With a screen editor you can scroll the page,
2  move the cursor, delete lines, insert characters,
   and more, while seeing the results of your edits
   as you make them.
3  Screen editors are very popular.
```

The number of logical lines (three) does not correspond to the number of visible lines (five) that you see on the screen. If the cursor were positioned on the *d* in the word *delete*, and you entered $, the cursor would move to the period following the word *them*. If you entered 0, the cursor would move back to the letter *m* in the word *move*, at the beginning of line two.

Movement by Text Blocks

You can also move the cursor by blocks of text, such as words, sentences, paragraphs, and so on:

w

    Move forward one word (alphanumeric characters make up words)

W

    Move forward one Word (whitespace separates words)

b

    Move backward one word (alphanumeric characters make up words)

B

    Move backward one Word (whitespace separates words)

G

    Go to a specific line

The w command moves the cursor forward one word at a time, counting symbols and punctuation as equivalent to words. The following line shows cursor movement by w:

    cursor, delete lines, insert characters,

You can also move forward by word, not counting symbols and punctuation, using the W command. (You can think of this as a "large" or "capital" *W*ord.)

Cursor movement using W looks like this:

    cursor, delete lines, insert characters,

To move backward by word, use the b command. Capital B allows you to move backward by word, not counting punctuation (by Word).

As mentioned previously, movement commands take numeric arguments; so, with either the w or b command you can multiply the movement with numbers. 2w moves forward two words; 5B moves back five words, not counting punctuation.

To move to a specific line, you can use the G command. Plain G goes to the end of the file, 1G goes to the top of the file, and 42G goes to line 42. This is described in more detail later, in the section "The G (Go To) Command" on page 52.

We discuss movement by sentences and by paragraphs in Chapter 3, "Moving Around in a Hurry". For now, practice using the cursor movement commands that you know, combining them with numeric multipliers.

Simple Edits

When you enter text in your file, it is rarely perfect. You find typos or want to improve on a phrase; sometimes your program has a bug. Once you enter text, you have to be able to change it, delete it, move it, or copy it. Figure 2-3 shows the kinds of edits you might want to make to a file. The edits are indicated by proofreading marks.


```
                screen                    seeing the
                                                        insert
    With a editor you can scroll the
    page, move the cursor, delete lines, (nisret)
    characters, and more, while results of
    your edits as you make them.                        e
    Since they allow you to make changes
    as you read through a file, much as
    you would edit a printed copy
    (screen editors are very popular)
```

Figure 2-3. Proofreading edits

In `vi` you can perform any of these edits with a few basic keystrokes: `i` for insert (which you've already seen); `a` for append; `c` for change; and `d` for delete. To move or copy text, you use pairs of commands. You move text with a `d` for "delete," then a `p` for "put"; you copy text with a `y` for "yank," then a `p` for "put." You may also use `x` to delete a single character and `r` to replace a single character. Some commands when doubled, such as `dd`, mean "apply the command to the entire line." Other commands when capitalized, such as `P`, mean "do the operation above the current line, instead of below it." Each type of edit is described in this section. Figure 2-4 shows the `vi` commands you use to make the edits marked in Figure 2-3.

Figure 2-4. Edits with vi commands

The text of this file is available for you to practice on; you can get it from the book's companion GitHub repository. See the section "Accessing the Files" on page 471 for more information.

Inserting New Text

You have already seen the insert command (i) used to enter text into a new file. You also use the insert command while editing existing text to add missing characters, words, and sentences. In the file *practice*, suppose you have the following sentence:

```
you can scroll
the page, move the cursor, delete
lines, and insert characters.
```

with the cursor positioned as shown. To insert *With a screen editor* at the beginning of the sentence, enter the following:

Keystrokes	Results
2k	you can scroll the page, move the cursor, delete lines, and insert characters. Move the cursor up two lines with the k command, to the line where you want to make the insertion.
iWith□a□	With a you can scroll the page, move the cursor, delete lines, and insert characters. Press i to enter insert mode and begin inserting text. The □ represents a space.

Keystrokes	Results
screen☐editor☐ [ESC]	With a screen editor█you can scroll the page, move the cursor, delete lines, and insert characters.
	Finish inserting text, and press [ESC] to end the insert and return to command mode.

Appending Text

You can append text at any place in your file with the append command, a. This works in almost the same way as i, except that text is inserted *after* the cursor rather than *before* the cursor. You may have noticed that when you press i to enter insert mode, the cursor doesn't move until after you enter some text. By contrast, when you press a to enter insert mode, the cursor moves one space to the right. When you enter text, it appears after the original cursor position.

Changing Text

You can replace any text in your file with the change command, c. To tell c how much text to change, you combine c with a movement command. In this way, a movement command serves as a *text object* for the c command to affect. For example, c can be used to change text from the cursor:

cw
> To the end of a word

c2b
> Back two words

c$
> To the end of the line

c0
> To the beginning of the line

After issuing a change command, you can replace the identified text with any amount of new text, with no characters at all, with one word, or with hundreds of lines. c, like i and a, leaves you in insert mode until you press the [ESC] key.

When the change affects only the current line, vi marks the end of the text to be changed with a $, so that you can see what part of the line is affected. If not in compatibility mode, Vim behaves differently; it simply removes the text to be changed and puts you into input mode.

Words

To change a word, combine the `c` (change) command with `w` for word. You can replace a word (`cw`) with a longer or shorter word (or any amount of text). `cw` can be thought of as "delete the word marked and insert new text until [ESC] is pressed."

Suppose you have the following line in your file *practice*:

```
    With an editor you can scroll the page,
```

and want to change *an* to *a screen*. You need to change only one word:

Keystrokes	Results
w	With a̲n editor you can scroll the page,
	Move with w to the place you want the edit to begin.
cw	With ▮editor you can scroll the page,
	Give the change word command. Vim deletes the *an* and goes into insert mode.
a screen[ESC]	With a scree̲n editor you can scroll the page,
	Type in the replacement text, and then press [ESC] to return to command mode.

`cw` also works on a portion of a word. For example, to change *spelling* to *spelled*, you can position the cursor on the *i*, type `cw`, then type *ed*, and finish with [ESC].

General Form of vi Commands

In the change commands we've mentioned up to this point, you may have noticed the following pattern:

  (*command*)(*text object*)

command is the change command `c`, and *text object* is a movement command (you don't type the parentheses). But `c` is not the only command that requires a text object. The `d` command (delete) and the `y` command (yank) follow this pattern as well.

Remember also that movement commands take numeric arguments, so numbers can be added to the text objects of `c`, `d`, and `y` commands. For example, `d2w` and `2dw` are commands to delete two words. With this in mind, you can see that most `vi` commands follow a general pattern:

  (*command*)(*number*)(*text object*)

or the equivalent form:

  (*number*)(*command*)(*text object*)

Here's how this works: *number* and *command* are optional. Without them, you simply have a movement command. If you add a *number*, you have a multiple movement.

On the other hand, you can combine a *command* (c, d, or y) with a *text object* to get an editing command.

When you realize how many combinations are possible in this way, Vim becomes a powerful editor indeed!

Lines

To replace the entire current line, use the special change command, cc. cc changes an entire line, replacing that line with any amount of text entered before pressing ESC. It doesn't matter where the cursor is located on the line; cc replaces the entire line of text.

With the original vi, a command like cw works differently from a command like cc. In using cw, the old text remains until you type over it, and any old text that is left over (up to the $) goes away when you press ESC. In using cc, though, the old text is wiped out first, leaving you a blank line on which to insert text.

The "type over" approach happens with any change command that affects less than a whole line, whereas the "blank line" approach happens with any change command that affects one or more lines.

With Vim (if not in compatibility mode), both commands simply delete the specified text and then enter input mode.

C replaces characters from the current cursor position to the end of the line. It has the same effect as combining c with the special end-of-line indicator $ (c$).

The commands cc and C are really shortcuts for other commands, so they don't follow the general form of vi commands, in that you can't specify a text object as the point at which the command ends. You'll see other shortcuts when we discuss the delete and yank commands.

Characters

One other replacement edit is given by the r command. r replaces a single character with another single character. You do *not* have to press ESC to return to command mode after making the edit. There is a misspelling in the following line:

```
Pith a screen editor you can scroll the page,
```

Only one letter needs to be corrected. You don't want to use cw in this instance because you would have to retype the entire word. Instead, use r to replace the single character at the cursor:

Keystrokes	Results
rW	▯ith a screen editor you can scroll the page,
	Give the replace command r, followed by the replacement character *W*.

Substituting text

Suppose you want to change just a few characters and not a whole word. The substitute command (s), by itself, replaces a single character. With a preceding count, you can replace that many characters. As with the change command (c), vi marks the last character of the text with a $ so that you can see how much text will be changed. Vim simply deletes the text and enters input mode. (You can think of s as being like r, but going into insert mode instead of directly replacing the specified character[s].)

The S command, as is usually the case with uppercase commands, lets you change whole lines. In contrast to the C command, which changes the rest of the line from the current cursor position, the S command deletes the entire line, no matter where the cursor is. The editor puts you in insert mode at the beginning of the line. A preceding count replaces that many lines. (S and cc are effectively equivalent.)

Both s and S put you in insert mode; when you are finished entering new text, press ESC.

The R command, like its lowercase counterpart, replaces text. The difference is that R simply enters *overstrike mode*. The characters you type replace what's on the screen, character by character, until you type ESC. If you're in the middle of a paragraph and you type R, you can overstrike a maximum of only one line; when you type ENTER, the editor opens a new line, effectively putting you in insert mode.

Changing Case

Changing the case of a letter is a special form of replacement. The tilde (~) command changes a lowercase letter to uppercase or an uppercase letter to lowercase. Position the cursor on the letter whose case you want to change, and type a ~. The case of the letter changes, and the cursor moves to the next character.

Provide a numeric prefix to change the case of multiple characters.

If you want to change the case of more than one line at a time, you must filter the text through a Unix command such as tr, as described in Chapter 7.

Deleting Text

You can also delete any text in your file with the delete command, d. Like the change command, the delete command requires a text object (the amount of text to be

operated on). You can delete by word (`dw`), by line (`dd` and `D`), or by other movement commands that you will learn later.

With all deletions, you move to where you want the edit to take place and then give the delete command (`d`) and the text object, such as `w` for word.

Words

Suppose you have the following text in the file:

```
Screen editors are are very popular,
since they allow you to make
changes as you read through a file.
```

with the cursor positioned as shown. You want to delete one *are* in the first line:

Keystrokes	Results
2w	Screen editors are are very popular, since they allow you to make changes as you read through a file.
	Move the cursor to where you want the edit to begin (*are*).
dw	Screen editors are very popular, since they allow you to make changes as you read through a file.
	Give the delete word command (dw) to delete the word *are*.

`dw` deletes a word beginning where the cursor is positioned. Notice that the space following the word is deleted as well.

`dw` can also be used to delete a portion of a word. In this example:

```
since they allowed you to make
```

you want to delete the *ed* from the end of *allowed*:

Keystrokes	Results
dw	since they allowyou to make
	Give the delete word command (dw) to delete the rest of the word, beginning with the position of the cursor.

`dw` always deletes the space before the next word on a line, but we don't want to do that in this example. To retain the space between words, use `de`, which deletes only to the end of a word. Typing `dE` deletes to the end of a word, including punctuation.[5]

You can also delete backward (`db`) or to the end or beginning of a line (`d$` or `d0`).

---

5 Robert P. J. Day points out that, unlike `dw` versus `de`, the commands `cw` and `ce` do the same thing.

Let's clarify the distinction between "words" and "Words." Suppose you have this text in your file:

```
This doesn't compute.
```

With the cursor at the start of the line, demonstrate the difference between `dw` and `dW` as follows:

Keystrokes	Results
w	`This doesn't compute.`
	Move the cursor to the *d*.
dw	`This t compute.`
	Delete the word under the cursor, up to but not including the punctuation.
u	`This doesn't compute.`
	Restore the line to what it was before.
dW	`This compute.`
	Delete the word under the cursor, up to the next whitespace character.

Lines

The `dd` command deletes the entire line that the cursor is on. `dd` does not delete part of a line. Like its complement, `cc`, `dd` is a special command. Using the same text as in the earlier example, with the cursor positioned on the first line as shown here:

```
Screen editors are very popular,
since they allow you to make
changes as you read through a file.
```

you can delete the first two lines:

Keystrokes	Results
2dd	`changes as you read through a file.`
	Give the command to delete two lines (2dd). Note that even though the cursor was not positioned on the beginning of the line, the entire line is deleted.

The `D` command deletes from the cursor position to the end of the line. (`D` is a shortcut for `d$`.) For example, with the cursor positioned as shown:

```
Screen editors are very popular,
since they allow you to make
changes as you read through a file.
```

you can delete the portion of the line under and to the right of the cursor:

Keystrokes	Results
D	Screen editors are very popular, since they allow you to make changes█
	Give the command to delete the portion of the line under and to the right of the cursor (D).

Characters

Often you want to delete only one or two characters. Just as r is a special change command to replace a single character, x is a special delete command to delete a single character. x deletes only the character the cursor is on. In the line here:

    zYou can move text by deleting text and then

you can delete the letter *z* by pressing x.[6] A capital X deletes the character before the cursor. Prefix either of these commands with a number to delete that number of characters. For example, 5x deletes the five characters under and to the right of the cursor. After using x or X you remain in command mode.

Problems with deletions

- *You've deleted the wrong text and you want to get it back.*

  There are several ways to recover deleted text. If you've just deleted something and you realize you want it back, simply type u to undo the last command (for example, a dd). This works only if you haven't given any further commands, since u undoes only the most recent command. Alternatively, a U restores the line to its pristine state, the way it was before *any* changes were applied to it.

  You can still recover a recent deletion, however, by using the p command, since vi saves the last nine deletions in nine numbered deletion registers. If you know, for example, that the third deletion back is the one you want to restore, type:

        "3p

  to "put" the contents of deletion register number three on the line below the cursor.

  This works only for a deleted *line*. Words, or a portion of a line, are not saved in a register. If you want to restore a deleted word or line fragment, and u won't work, use the p command by itself. This restores whatever you've last deleted.

  Note that Vim supports "infinite" undo, which makes life much easier. See the section "Extended Undo" on page 180 for more information.

---

6 The mnemonic for x is that it is supposedly like "x-ing out" mistakes with a typewriter. Of course, who uses a typewriter anymore?

Undo undoes the last operation, whatever it was. Deleting two words by typing dw twice is two operations: u restores only the last deleted word. However, deleting two words by typing 2dw is a single operation; u in this case restores both deleted words.

Moving Text

Each time you delete a text block, that deletion is saved in a special, unnamed place, which we will call the *deletion register*.[7] The register's contents are overwritten upon each new deletion.

In vi, you move text by deleting it and then placing that deleted text elsewhere in the file, like a "cut and paste." After deleting the text to be moved, move to another position in your file and use the put command (p) to place that text in the new position. You can move any block of text, although moving is more useful with lines than with words.

The put command (p) puts the text that is in the deletion register *after* the cursor position. The uppercase version of the command, P, puts the text *before* the cursor. If you delete one or more lines, p puts the deleted text on a new line (or lines) below the cursor, whereas P puts the text on a new line (or lines) above the cursor. If you delete less than an entire line, p puts the deleted text into the current line, after the cursor.

Suppose in your file *practice* you have the text:

```
You can move text by deleting it and then,
like a "cut and paste,"
placing the deleted text elsewhere in the file.
each time you delete a text block.
```

and you want to move the second line, *like a "cut and paste,"* below the third line. Using delete, you can make this edit:

Keystrokes	Results
dd	You can move text by deleting it and then, placing the deleted text elsewhere in the file. each time you delete a text block. With the cursor on the second line, delete that line. The text is placed in the deletion register.

---

7 Older vi documentation calls this the *deletion buffer*. We use Vim's term, *register*, to avoid confusion with the buffers that hold file contents.

Keystrokes	Results
| p | ```
You can move text by deleting it and then,
placing that deleted text elsewhere in the file.
Like a "cut and paste"
each time you delete a text block.
``` |
| | Give the put command, p, to restore the deleted line at the next line below the cursor. To finish reordering this sentence, you would also have to change the capitalization and punctuation (with r) to match the new structure. |

> Once you delete text, you must restore it before the next change command or delete command. If you make another edit that saves text to the deletion register, your previously deleted text will be lost. You can repeat the put over and over, so long as you don't make a new edit. In the section "Making Use of Registers" on page 60, you will learn how to save text you delete in a named register so that you can retrieve it later.

Transposing two letters

You can use xp (delete character and put after cursor) to transpose two letters. For example, in the word *mvoe*, the letters *vo* are transposed (reversed). To correct a transposition, place the cursor on *v* and press x, then p. By coincidence, the word *transpose* helps you remember the sequence xp; x stands for *trans*, and p stands for *pose*.

There is no command to transpose words. The section "Using the map Command" on page 124 discusses a short sequence of commands that transposes two words.

Copying Text

Often you can save editing time (and keystrokes) by copying a part of your file to use in other places. With the two commands y (for yank) and p (for put), you can copy any amount of text and put that copied text in another place in the file. A yank command copies the selected text into the deletion register, where it is held until another yank (or deletion) occurs. You can then place this copy elsewhere in the file with the put command.

As with change and delete, the yank command can be combined with any movement command (yw, y$, y0, 4yy). Yank is most frequently used with a line (or more) of text, because to yank and put a word usually takes longer than simply to insert the word.

The shortcut yy operates on an entire line, just as dd and cc do. But the shortcut Y, for some reason, does not operate the way D and C do. Instead of yanking from the current position to the end of the line, Y yanks the whole line; that is, Y does the same thing as yy. (Use y$ to yank from the current position to the end of the line.)

Suppose you have in your file *practice* the following text:

```
With a screen editor you can
scroll the page.
move the cursor.
delete lines.
```

You want to make three complete sentences, beginning each with *With a screen editor you can*. Instead of moving through the file and making this edit over and over, you can use a yank and put to copy the text to be added:

Keystrokes	Results

yy	```With a screen editor you can```
	```scroll the page.```
	```move the cursor.```
	```delete lines.```
	
	Yank the line of text that you want to copy into the register. The cursor can be anywhere on the line you want to yank (or on the first line of a series of lines).

2j	```With a screen editor you can```
	```scroll the page.```
	```move the cursor.```
	```delete lines.```
	
	Move the cursor to where you want to put the yanked text.

P	```With a screen editor you can```
	```scroll the page.```
	```With a screen editor you can```
	```move the cursor.```
	```delete lines.```
	
	Put the yanked text above the cursor line with P.

jp	```With a screen editor you can```
	```scroll the page.```
	```With a screen editor you can```
	```move the cursor.```
	```With a screen editor you can```
	```delete lines.```
	
	Move the cursor down a line. Then put the yanked text below the cursor's line with p.

Yanking uses the same register as deleting. Each new deletion or yank replaces the previous contents of the deletion register. As we'll see in "Making Use of Registers" on page 60, up to nine previous yanks or deletions can be recalled with put commands. You can also yank or delete directly into up to 26 named registers, which allows you to juggle multiple text blocks at once.

Repeating or Undoing Your Last Command

Each edit command that you give is stored in a temporary register until you give the next command. For example, if you insert *the* after a word in your file, the command used to insert the text, along with the text that you entered, is temporarily saved.

Repeat

Any time you make the same editing command over and over, you can save time by duplicating it with the repeat command, the period (.). Position the cursor where you want to repeat the editing command, and type a period.

Suppose you have just the following lines in your file:

```
With a screen editor you can
scroll the page.
With a screen editor you can
move the cursor.
```

You can delete one line, and then, to delete another line, simply type a period:

Keystrokes	Results
dd	With a screen editor you can scroll the page. move the cursor.
	Delete a line with the command dd.
.	With a screen editor you can scroll the page.
	Repeat the deletion.

Undo

As mentioned earlier, you can undo your last command if you make an error. Simply press u. The cursor need not be on the line where the original edit was made.

To continue the previous example, showing deletion of lines in the file *practice*:

Keystrokes	Results
u	With a screen editor you can scroll the page. move the cursor.
	u undoes the last command and restores the deleted line.

In vi, U, the uppercase version of u, undoes all edits on a single line, *as long as the cursor remains on that line*. Once you move off a line, you can no longer use U. Vim does not have this restriction.

Note that you can undo your last undo with u, toggling between two versions of text. u also undoes U, and U undoes any changes to a line, including those made with u.

If you're working with Vim, it's likely that undo works differently, simply undoing successive changes. Vim lets you use CTRL-R to "redo" an undone operation. Combined with infinite undo, you can move backward and forward through the

history of changes to your file. See the section for more information.

The fact that u can undo itself leads to a nifty way to get around in a file. If you ever want to get back to the site of your last edit, simply undo it. You will pop back to the appropriate line. When you undo the undo, you'll stay on that line.

More Ways to Insert Text

You have inserted text before the cursor with the following sequence:

`itext to be inserted` [ESC]

You've also inserted text after the cursor with the `a` command. Here are some other insert commands for inserting text at different positions relative to the cursor (some were discussed earlier):

A

    Append text to the end of the current line.

I

    Insert text at the beginning of the current line.

o *(lowercase letter "o")*
    Open an empty line below the cursor for text.

O *(uppercase letter "o")*
    Open an empty line above the cursor for text.

s

    Delete the character at the cursor and substitute text.

S

    Delete the current line and substitute text.

R

    Starting at the cursor, overstrike existing characters with new characters.

All of these commands place you in insert mode. After inserting text, remember to press [ESC] to return to command mode.

A (append) and I (insert) save you from having to move your cursor to the end or beginning of the line before invoking insert mode. (The A command saves one keystroke over `$a`. Although one keystroke might not seem like much of a saving, the more adept—and impatient—an editor you become, the more keystrokes you will want to omit.)

o and O (open) save you from having to insert a carriage return. You can type these commands from anywhere within the line.

s and S (substitute) allow you to delete a character or a whole line and replace the deletion with any amount of new text. s is the equivalent of the two-stroke command c SPACE , and S is the same as cc. One of the best uses for s is to change one character to several characters.

R ("large" replace) is useful when you want to start changing text, but you don't know exactly how much. For example, instead of guessing whether to say 3cw or 4cw, just type R and then enter your replacement text.

Numeric Arguments for Insert Commands

Except for o and O, the insert commands just listed (plus i and a) take numeric prefixes. With numeric prefixes, you might use the commands i, I, a, and A to insert a row of underlines or alternating characters. For example, typing 50i* ESC inserts 50 asterisks, and typing 25a*- ESC appends 50 characters (25 pairs of asterisk and hyphen). It's better to repeat only a small string of characters.

With a numeric prefix, r replaces that number of characters with a repeated instance of a single character. For example, in C or C++ code, to change || to &&, you would place the cursor on the first pipe character and type 2r&.

You can use a numeric prefix with S to substitute several lines. It's quicker and more flexible, though, to use c with a movement command.

A good case for using the s command with a numeric prefix is when you want to change a few characters in the middle of a word. Typing r wouldn't be correct, and typing cw would change too much text. Using s with a numeric prefix is about the same as typing R.

There are other combinations of commands that work naturally together. For example, ea is useful for appending new text to the end of a word. It helps to train yourself to recognize such useful combinations so that they become automatic.

Joining Two Lines with J

Sometimes while editing a file you end up with a series of short lines that are difficult to scan.

Suppose your file *practice* reads as follows:

```
With a
screen editor
you can
scroll the page, move the cursor
```

When you want to merge two lines into one, position the cursor anywhere on the first line and press SHIFT-J to join the two lines:

Keystrokes	Results
J	With a█screen editor you can scroll the page, move the cursor
	J joins the line the cursor is on with the line below.
.	With a screen editor█you can scroll the page, move the cursor
	Repeat the last command (J) with the . to join the next line with the current line.

Using a numeric argument with `J` joins that number of consecutive lines. Both `1J` and `2J` join the current line with the line following it. A numeric argument of three or more joins that many lines, including the line where the cursor is. In the example here, you could have joined the first three lines by using the command `3J`.

Problems with vi Commands

- *When you type commands, text jumps around on the screen and nothing works the way it's supposed to.*

  Make sure you're not typing the `J` command when you mean `j`.

  You may have hit the CAPS LOCK key without noticing it; `vi` and Vim are case sensitive—that is, uppercase commands (e.g., `I`, `A`, `J`) are different from lowercase commands (`i`, `a`, `j`)—and if you hit this key, all your commands are interpreted not as lowercase but as uppercase commands. Press the CAPS LOCK key again to return to lowercase, press ESC to ensure that you are in command mode, and then type either `U` to restore the last line changed or `u` to undo the last command. You'll probably also have to do some additional editing to fully restore the garbled part of your file.

Mode Indicators

As you know by now, the editor has two modes—command mode and insert mode. Usually, you can't tell by looking at the screen which mode you're in. Furthermore, it's often useful to know where in the file you are, without having to use the CTRL-G or `ex :.=` commands.

Two options address these issues: `showmode` and `ruler`. Vim has both, while the "Heirloom" and Solaris `/usr/xpg7/bin` versions of `vi` have the `showmode` option.

Table 2-1 lists the special features in each editor.

Table 2-1. Position and mode indicators

Editor	With ruler, displays	With showmode, displays
vi	N/A	Separate mode indicators for open, input, insert, append, change, replace, replace one character, and substitute modes
Vim	Row and column	Insert, replace, and visual mode indicators

Review of Basic vi Commands

Table 2-2 presents a few of the commands you can perform by combining the commands c, d, and y with various text objects. The last two rows show additional commands for editing. Tables 2-3 and 2-4 list some other basic commands. Table 2-5 summarizes the rest of the commands described in this chapter.

Table 2-2. Edit commands

Text object	Change	Delete	Copy (yank)
One word	cw	dw	yw
Two words, whitespace separated	2cW or c2W	2dW or d2W	2yW or y2W
Three words back	3cb or c3b	3db or d3b	3yb or y3b
One line	cc	dd	yy or Y
To end of line	c$ or C	d$ or D	y$
To beginning of line	c0	d0	y0
Single character	r	x or X	yl or yh
Five characters	5s	5x	5yl

Table 2-3. Movement

Movement	Commands
←, ↓, ↑, →	h, j, k, l
←, ↓, ↑, →	BACKSPACE, CTRL-N and ENTER, CTRL-P, space bar
To first character of next line	+
To first character of previous line	-
To end of word	e or E
Forward by word	w or W
Backward by word	b or B
To end of line	$
To beginning of line	0
To a particular line	G

Table 2-4. Other operations

Operations	Commands
Place text from register	p or P
Start vi, open file if specified	vi *file*
Start Vim, open file if specified	vim *file*
Save edits, quit file	ZZ
No saving of edits, quit file	:q! ENTER

Table 2-5. Text creation, deletion, and manipulation commands

Editing action	Command
Insert text at current position	i
Insert text at beginning of line	I
Append text at current position	a
Append text to end of line	A
Open new line below cursor for new text	o
Open new line above cursor for new text	O
Put deleted text after cursor or below current line	p
Put deleted text before cursor or above current line	P
Replace the character under the cursor	r
Overstrike existing characters with new text	R
Delete the current character and enter insert mode	s
Delete line and substitute text	S
Delete the character under the cursor	x
Delete the character in front of the cursor	X
Join current and next line	J
Toggle case	~
Repeat last action	.
Undo last change	u
Restore line to original state	U

You can get by in vi and Vim using only the commands listed in these tables. However, to harness the real power of the editor (and increase your own productivity), you will need more tools. The following chapters describe those tools.

Moving Around in a Hurry

You will not, of course, just create new files. You'll spend a lot of your time editing existing files. You rarely want to simply open to the first line in the file and move through it line by line; you sometimes want to get to a specific place in a file and start working.

All edits start with you moving the cursor to where you want to begin the edit (or, with `ex` line editor commands, by identifying the line numbers to be edited). This chapter shows you how to think about movement in a variety of ways (by screens, by text, by patterns, or by line numbers). There are many ways to move around in `vi` and Vim, since editing speed depends on getting to your destination with only a few keystrokes.

This chapter covers:

- Movement by screens
- Movement by text blocks
- Movement by searches for patterns
- Movement by line number

Movement by Screens

When you read a book, you think of "places" in the book in terms of pages: the page where you stopped reading, or the page number in an index. You don't have this convenience when you're editing files. Some files take up only a few lines, and you can see the whole file at once. But many files have hundreds (or thousands!) of lines.

You can think of a file as text on a long roll of paper. The screen is a window of (often) 24 lines of text on that long roll.[1]

In insert mode, as you fill up the screen with text, you will end up typing on the bottom line of the screen. When you reach the end and press ENTER, the top line rolls out of sight, and a blank line appears on the bottom of the screen for new text. This is called *scrolling*.

In command mode, you can move through a file to see any text in it by scrolling the screen ahead or back. And since cursor movements can be multiplied by numeric prefixes, you can move quickly to anywhere in your file.

Scrolling the Screen

There are `vi` commands to scroll forward and backward through the file by full and half screens:

`^F`

> Scroll forward one screen.

`^B`

> Scroll backward one screen.

`^D`

> Scroll forward a half screen (down).

`^U`

> Scroll backward a half screen (up).

> In this list of commands, the `^` symbol represents the CTRL key. So `^F` means to hold down the CTRL key and press the SHIFT-F key simultaneously.

There are also commands to scroll the screen up one line (`^E`) and down one line (`^Y`). However, these two commands do not send the cursor to the beginning of the line. The cursor remains at the same point in the line as it was when the command was issued.

---

1 The editor knows how big your screen is, even if you adjust the size of your terminal emulator's window while editing.

Repositioning the Screen with z

If you want to scroll the screen up or down, but you want the cursor to remain on the line where you left it, use the z command:

z ENTER *and* z+ ENTER
>   Move the current line to the top of the screen and scroll.

z.
>   Move the current line to the center of the screen and scroll.

z-
>   Move the current line to the bottom of the screen and scroll.

With the z command, using a numeric prefix as a multiplier makes no sense. (After all, you would need to reposition the cursor to the top of the screen only once. Repeating the same z command wouldn't move anything.) Instead, z understands a numeric prefix as a line number that it uses in place of the current line. For example, z ENTER moves the current line to the top of the screen, but 200z ENTER moves line 200 to the top of the screen.

> Some GNU/Linux distributions come with an */etc/vimrc* file that sets the Vim option scrolloff ("scroll offset") to a nonzero value (typically five).[2] Others use a file */usr/share/vim/vimXX/ defaults.vim* where *XX* is the Vim version. Setting scrolloff to a nonzero value causes Vim to always provide that many lines of context above and below the cursor. Thus, if you type z ENTER to move the current line to the top of the screen, but the current line only moves to a few lines below the top of the screen, you'll know why.
>
> This option also affects the H and L commands (see "Movement Within a Screen" on page 44), and maybe others.
>
> You can cancel the effect of such a default setting by explicitly setting scrolloff to zero in your personal *.vimrc* file. (For more information on this file, see the section "Customizing vi and Vim" on page 114, and the section "System and User Configuration Files" on page 171.)

Redrawing the Screen

If you're using vi or Vim in a terminal window, messages from your computer system may display on your screen while you're editing. (This can happen particularly if

---

2 Thanks to Robert P. J. Day for noticing this on his system and telling us about it.

you're logged in to a remote server system.) These messages don't become part of your editing buffer, but they do interfere with your work. When system messages appear on your screen, you need to redisplay, or redraw, the screen.

Whenever you scroll, you redraw part (or all) of the screen, so you can always get rid of unwanted messages by scrolling them off the screen and then returning to your previous position. But you can also redraw the screen without scrolling, by typing CTRL-L.

Movement Within a Screen

You can also keep your current screen or view of the file and move around within the screen using these commands:

H

> Move to home—the first character on the top line on the screen.

M

> Move to the first character on the middle line on the screen.

L

> Move to the first character on the last line on the screen.

n H

> Move to the first character on the line *n* lines below the top line.

n L

> Move to the first character on the line *n* lines above the last line.

H moves the cursor from anywhere on the screen to the first or "home" line. M moves to the middle line, L to the last; to move to the line below the first line, use 2H:

Keystrokes	Results
| L | ```
With a screen editor you can
scroll the page, move the cursor,
delete lines, insert characters, and more,
while seeing the results of your
edits as you make them.
Screen editors are very popular,
since they allow you to make changes
as you read through a file.
```<br><br>Move to the last line of the screen with the L command. |

Keystrokes	Results
| 2H | ```
With a screen editor you can
scroll the page, move the cursor,
delete lines, insert characters, and more,
while seeing the results of your
edits as you make them.
Screen editors are very popular,
since they allow you to make changes
as you read through a file.
``` |
| | Move to the second line of the screen with the 2H command. (H alone moves to the top line of the screen.) |

Movement by Line

Within the current screen there are also commands to move by line. You've already seen j and k. You can also use:

ENTER
> Move to the first nonblank character of the next line.

+
> Move to the first nonblank character of the next line. (Same as ENTER.)

-
> Move to the first nonblank character of the previous line.

These three commands move down or up to the first *character* of the line, ignoring any spaces or tabs. j and k, by contrast, move the cursor down or up to the first position of a line, even if that position is blank (and assuming that the cursor started at the first position).

Movement on the current line

Don't forget that h and l move the cursor to the left and right, and that 0 (zero) and $ move the cursor to the beginning or end of the line. You can also use the following commands:

^
> Move to the first nonblank character of the current line.

n |
> Move to the character in column *n* of the current line, or to the end of the line if *n* is greater than the number of characters on the line.[3]

---

3 Why the |? It comes from the similar use of | as a motion command in the troff formatter.

As with the line movement commands shown earlier, ^ moves to the first *character* of the line, ignoring any spaces or tabs. 0, by contrast, moves to the first position of the line, even if that position is blank.

Movement by Text Blocks

Another way that you can think of moving through a vi file is by text blocks—words, sentences, paragraphs, or sections.

You have already learned to move forward and backward by word (w, W, b, or B). In addition, you can use these commands:

e

    Move to the end of the current word (punctuation and whitespace separate words).

E

    Move to the end of the current word (whitespace separates words).

(

    Move to the beginning of the current sentence.

)

    Move to the beginning of the next sentence.

{

    Move to the beginning of the current paragraph.

}

    Move to the beginning of the next paragraph.

[[

    Move to the beginning of the current section.

]]

    Move to the beginning of the next section.

To find the end of a sentence, vi and Vim look for one of these punctuation marks: ?, ., or !. vi locates the end of a sentence when the punctuation is followed by at least two spaces or when it appears as the last nonblank character on a line. If you have left only a single space following a period, or if the sentence ends with a quotation mark, vi won't recognize the sentence. However, Vim is not quite as old-fashioned, requiring only a single space following the terminating punctuation.

A paragraph is defined as text up to the next blank line, or up to one of the default paragraph macros (`.IP`, `.PP`, `.LP`, or `.QP`) from the `troff` MS macro package.[4] Similarly, a section is defined as text up to the next default section macro (`.NH`, `.SH`, `.H 1`, or `.HU`). The macros that are recognized as paragraph or section separators can be customized with the `:set` command, as described in Chapter 7, "Advanced Editing".

Remember that you can combine numbers with movement. For example, `3)` moves ahead three sentences. Also remember that you can edit using movement commands: `d)` deletes to the end of the current sentence; `2y}` copies (yanks) two paragraphs ahead.

Remember too that you can use motion commands with editing commands such as `cw` and `ce`. Robert P. J. Day points out that, interestingly, "while `w` and `e` are slightly different movement commands, the change commands `cw` and `ce` do exactly the same thing."

Movement by Searches

One of the most useful ways to move around quickly in a large file is by searching for text, or, more properly, for a *pattern* of characters. Sometimes a search can be performed to find a misspelled word or to find each occurrence of a variable in a program.

The search command is the special character `/` (slash). When you enter a slash, it appears on the bottom line of the screen; you then type in the *pattern* that you want to find: `/pattern`.

A pattern can be a whole word or any other sequence of characters (called a "character string"). For example, if you search for the characters *red*, you will match *red* as a whole word, but you'll also match occur*red*. If you include a space before or after *pattern*, the spaces are treated as part of the text to be matched. As with all bottom-line commands, press ENTER to finish.

`vi` and Vim, like all other Unix editors, have a special pattern-matching language that allows you to look for variable text patterns—for example, any word beginning with a capital letter, or the word *The* at the beginning of a line. We'll talk about this more powerful pattern-matching syntax in Chapter 6, "Global Replacement". For right now, simply think of a pattern as a word or phrase.

---

4 This is less useful than it used to be. `troff` is still used, but not as much as when Unix was young.

The editor begins the search at the cursor and searches forward, wrapping around to the start of the file if necessary. The cursor moves to the first occurrence of the pattern. If there is no match, the message "Pattern not found" is shown on the status line.[5]

Using the file *practice*, here's how to move the cursor by searches:

Keystrokes	Results
/edits `ENTER`	```With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your ▊dits as you make them.```
	Search for the pattern *edits*. The cursor moves directly to the matching text. Note that you do not type a space after *edits* before hitting `ENTER`.
/scr `ENTER`	```With a ▊creen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them.```
	Search for the pattern *scr*. The search wraps around to the front of the file.

Note that you can give any combination of characters; a search does not have to be for a complete word.

To search backward, type a ? instead of a /:

    ?pattern

In both cases, the search wraps around to the beginning or end of the file, if necessary.

Repeating Searches

The last pattern that you searched for stays available throughout your editing session. After a search, instead of repeating your original keystrokes, you can use a `vi` command to search again for the last pattern:

n

> Repeat the search in the same direction.

N

> Repeat the search in the opposite direction.

---

[5] The exact messages vary with different editor versions, but their meanings are the same. In general, we won't bother noting everywhere that the text of a message may be different; in all cases the information conveyed is the same.

/ ENTER
> Repeat the search forward.

? ENTER
> Repeat the search backward.

Since the last pattern stays available, you can search for a pattern, do some work, and then search again for the same pattern without retyping it by using n, N, /, or ?. The direction of your search (/ is forward, ? is backward) is displayed at the bottom left of the screen. Vim goes beyond vi by putting the search text into the command line and letting you scroll through a saved history of search commands, using the up and down arrow keys. The section "Introducing the history windows" on page 341 discusses how to take full advantage of the saved search text history.

To continue with the previous example, since the pattern *scr* is still available for search, you can do the following:

Keystrokes	Results
| n | ```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters, and more, while seeing the
results of your edits as you make them.
``` |
| | Move to the next instance of the pattern *scr* (from *scr*een to *scr*oll) with the n (next) command. |
| ?you ENTER | ```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters, and more, while seeing the
results of your edits as you make them.
``` |
| | Search backward with ? from the cursor to the first occurrence of *you*. You need to press ENTER after typing the pattern. |
| N | ```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters, and more, while seeing the
results of your edits as you make them.
``` |
| | Repeat the previous search for *you* but in the opposite direction (forward). |

Sometimes you want to find a word only if it is further ahead; you don't want the search to wrap around earlier in the file. There is an option, wrapscan, that controls whether searches wrap. You can disable wrapping like this:

```
:set nowrapscan
```

When nowrapscan is set and a forward search fails, vi's status line displays the following message:

```
Address search hit BOTTOM without matching pattern
```

Vim displays this message:

```
E385: search hit BOTTOM without match for: foo
```

When `nowrapscan` is set and a backward search fails, the messages display "TOP" instead of "BOTTOM."

Changing through searching

You can combine the `/` and `?` search operators with the commands that change text, such as `c` and `d`. Continuing with the previous example:

Keystrokes	Results
d?move ENTER	With a screen editor you can scroll the page, █our edits as you make them.
	Delete from before the cursor up to and through the word *move*.

Note how the deletion occurs on a character basis, and whole lines are not deleted.

This section has given you only the barest introduction to searching for patterns. Chapter 6 teaches you more about pattern matching and its use in making global changes to a file.

Current Line Searches

There are also miniature versions of the search commands that operate within the current line. The command `fx` moves the cursor to the next instance of the character *x* (where *x* stands for any character). The command `tx` moves the cursor to the character *before* the next instance of *x*. (`f` is short for *find*; `t` is short for *to*, meaning "up to.") Semicolons can then be used repeatedly to "find" your way along.

The inline search commands are summarized here. None of these commands move the cursor to the next line:

`fx`

Find (move cursor to) the next occurrence of *x* in the line, where *x* stands for any character.

`Fx`

Find (move cursor to) the previous occurrence of *x* in the line.

`tx`

Find (move cursor to) the character *before* the next occurrence of *x* in the line.

`Tx`

Find (move cursor to) the character *after* the previous occurrence of *x* in the line.

`;`

Repeat the previous find command in the same direction.

, *(comma)*
   Repeat the previous find command in the opposite direction.

With any of these commands, a numeric prefix *n* locates the *n*th occurrence.

Suppose you are editing in *practice*, on this line:

```
With a screen editor you can scroll the
```

You can find occurrences of the letter *o* as follows:

Keystrokes	Results
fo	With a screen edit**o**r you can scroll the
	Find the first occurrence of *o* in your current line with f.
;	With a screen editor y**o**u can scroll the
	Move to the next occurrence of *o* with the ; command (find next *o*).

d f*x* deletes up to and including the named character *x*. This command is useful in deleting or yanking partial lines. You might need to use d f*x* instead of dw if there are symbols or punctuation within the line that makes counting words difficult. The t command works just like f, except that it positions the cursor before the character searched for. For example, the command ct. could be used to change text up to the end of a sentence, leaving the period.

Movement by Line Number

Lines in a file are numbered sequentially, and you can move through a file by specifying line numbers.

Line numbers are useful for identifying the beginning and end of large blocks of text you want to edit. Line numbers are also useful for programmers, since compiler error messages refer to line numbers. Finally, line numbers are used by ex commands, which you will learn in the following chapters.

If you are going to move by line numbers, you must have a way to identify them. Line numbers can be displayed on the screen using the :set nu option described in Chapter 5. You can also display the current line number on the bottom of the screen.

The command CTRL-G causes vi to display the following at the bottom of your screen: the current line number, the total number of lines in the file, and what percentage of the total the present line number represents. For example, for the file *practice*, CTRL-G might display:

```
"practice" line 3 of 6 --50%--
```

Vim provides more information:

```
"practice" 4 lines --75%--                          3,23          All
```

The next-to-last field is the cursor position (line 3, character 23). In a larger file, the final field becomes a percentage indicating how far along in the file you are.

CTRL-G is useful either for displaying the line number to use in a command or for orienting yourself if you have been distracted from your editing session.

If you have modified the file but not yet written it out, [Modified] appears in the status line after the filename.

The G (Go To) Command

You can use line numbers to move the cursor through a file. The G (go to) command uses a line number as a numeric argument and moves directly to that line. For instance, 44G moves the cursor to the beginning of line 44. G without a line number moves the cursor to the last line of the file.

Typing two backquotes (``) returns you to your original position (the position where you issued the last G command), unless you have done some edits in the meantime. If you have made an edit and then moved the cursor using some command other than G, `` returns the cursor to the site of your last edit. If you have issued a search command (/ or ?), `` returns the cursor to its position when you started the search. A pair of apostrophes ('') works much like two backquotes, except that it returns the cursor to the beginning of the line instead of the exact position on that line where your cursor had been.

The total number of lines shown with CTRL-G can give you a rough idea of how many lines to move. If you are on line 10 of a 1,000-line file:

```
"practice" 1000 lines --1%--                        10,1          1%
```

and you know that you want to begin editing near the end of that file, you could give an approximation of your destination with 800G.

Movement by line number is a tool that can move you quickly from place to place through a large file.

Review of vi Motion Commands

Table 3-1 summarizes the commands covered in this chapter.

Table 3-1. Movement commands

Movement	Command
Scroll forward one screen	^F
Scroll backward one screen	^B
Scroll forward a half screen	^D
Scroll backward a half screen	^U
Scroll forward one line	^E
Scroll backward one line	^Y
Move current line to top of screen and scroll	z [ENTER]
Move current line to center of screen and scroll	z.
Move current line to bottom of screen and scroll	z-
Redraw the screen	^L
Move to home—the top line of screen	H
Move to middle line of screen	M
Move to bottom line of screen	L
Move to first character of next line	[ENTER]
Move to first character of next line	+
Move to first character of previous line	-
Move to first nonblank character of current line	^
Move to column *n* of current line	*n*	
Move to end of word	e
Move to end of word (ignore punctuation)	E
Move to beginning of current sentence	(
Move to beginning of next sentence	)
Move to beginning of current paragraph	{
Move to beginning of next paragraph	}
Move to beginning of current section	[[
Move to beginning of next section	]]
Search forward for pattern	*/pattern* [ENTER]
Search backward for pattern	*?pattern* [ENTER]
Repeat last search	n
Repeat last search in opposite direction	N
Repeat last search forward	/
Repeat last search backward	?
Move to next occurrence of *x* in current line	f*x*
Move to previous occurrence of *x* in current line	F*x*
Move to just before next occurrence of *x* in current line	t*x*

Movement	Command
Move to just after previous occurrence of *x* in current line	T*x*
Repeat previous find command in same direction	;
Repeat previous find command in opposite direction	,
Go to given line *n*	*n* G
Go to end of file	G
Return to previous mark or context	``
Return to beginning of line containing previous mark	' '
Show current line (not a movement command)	^G

Beyond the Basics

You have already been introduced to the basic editing commands, `i`, `a`, `c`, `d`, and `y`. This chapter expands on what you already know about editing. It covers:

- Descriptions of additional editing facilities, with a review of the general command form
- `vi` and Vim command-line options, including different ways to open a file for editing
- Making use of registers that store yanks and deletions
- Marking your place in a file
- Other advanced edits

More Command Combinations

In Chapter 2, "Simple Editing", you learned the edit commands `c`, `d`, and `y`, as well as how to combine them with movements and numbers (such as `2cw` or `4dd`). In Chapter 3, "Moving Around in a Hurry", you added many more movement commands to your repertoire. Although the fact that you can combine edit commands with movement is not a new concept to you, Table 4-1 presents several additional editing options you have not seen before.

Table 4-1. More editing commands

Change	Delete	Copy	From cursor to...
cH	dH	yH	Top of screen
cL	dL	yL	Bottom of screen
c+	d+	y+	Next line
c5|	d5|	y5|	Column 5 of current line
2c)	2d)	2y)	Second sentence following
c{	d{	y{	Previous paragraph
c/pattern	d/pattern	y/pattern	*Pattern*
cn	dn	yn	Next *pattern*
cG	dG	yG	End of file
c13G	d13G	y13G	Line number 13

Notice how all of the sequences in Table 4-1 follow one of two general patterns:

> (*command*)(*number*)(*text object*)

or:

> (*number*)(*command*)(*text object*)

number is the optional numeric argument. *command* in this case is one of c, d, or y. *text object* is a movement command.

The general form of a vi command is discussed in Chapter 2. You may wish to review Tables 2-2 and 2-3 as well.

Options When Starting vi and Vim

So far, you have invoked the editor from the shell with the command:

> $ **vi** *file*

or with

> $ **vim** *file*

There are other options to the vim command that can be helpful. You can open a file directly to a specific line number or pattern. You can also open a file in read-only mode. Another option recovers all changes to a file that you were editing when the system crashed.

The options described in the following section apply both to vi and to Vim.

Advancing to a Specific Place

When you begin editing an existing file, you can call the file in and, then move to the first occurrence of a pattern or to a specific line number. You can also specify your first movement by search or by line number right on the command line. You do this using `-c command`; for backward compatibility with earlier versions of `vi`, you may also use `+command`:

`$ vim -c n file`
> Open *file* at line number *n*.

`$ vim -c /pattern file`
> Open *file* at the first occurrence of *pattern*.

`$ vim + file`
> Open *file* at the last line.

In the file *practice*, to open the file and advance directly to the line containing the word *Screen*, enter the following:

Keystrokes	Results

`$ vim -c /Screen practice`	`With a screen editor you can scroll`
	`the page, move the cursor, delete`
	`lines, and insert characters, while`
	`seeing the results of your edits as`
	`you make them. Screen editors are`
	`very popular, since they allow you`
	`to make changes as you read`

	Give the `vim` command with the option `-c /pattern` to go directly to the line containing *Screen*.

As you see in this example, your search pattern is not necessarily positioned at the top of the screen. Interestingly, the cursor is placed on the first character of the line and not on the first character of the matching text! If you include spaces in the *pattern*, you must enclose the whole pattern within single or double quotes:[1]

    -c /"you make"

or escape the space with a backslash:

    -c /you\ make

In addition, if you want to use the general pattern-matching syntax described in Chapter 6, "Global Replacement", you may need to protect one or more special characters from interpretation by the shell with either single quotes or backslashes.

---

1 It is the shell that imposes the quoting requirement, not the editor.

Using `-c /pattern` is helpful if you have to leave an editing session before you're finished. You can mark your place by inserting a pattern such as ZZZ or HERE. Then, when you return to the file, all you have to remember is /ZZZ or /HERE.

After the editor opens your file and does a pattern search for the pattern you gave it with `-c`, you can continue to the next occurrence of that pattern simply by using n.

> Normally, when you're editing in vi and Vim, the `wrapscan` option is enabled. If you've customized your environment so that `wrapscan` is always disabled (see the section "Repeating Searches" on page 48), you might not be able to use `-c /pattern`. If you try to open a file this way, the editor opens the file at the last line and displays the message, "Address search hit BOTTOM without matching pattern." The message will likely vary among different versions of vi and Vim.

Read-Only Mode

There will be times when you want to look at a file but want to protect that file from inadvertent keystrokes and changes. (You might want to call in a lengthy file to practice vi movements, or you might want to scroll through a command file or program.) You can enter a file in read-only mode and use all the regular movement commands, but you won't be able to change the file.

To look at a file in read-only mode, enter either:

    $ **vim -R** *file*

or:

    $ **view** *file*

(The view command, like Vim, can use any of the command-line options for advancing to a specific place in the file.[2]) If you do decide to make some edits to the file, you can override read-only mode by adding an exclamation point to the `write` command:

    :w!

or:

    :wq!

Note that if you edit a file for which you do not have write permission, you will also be in read-only mode. In that case, if you own the file, a :w! or :wq! will still work; vi temporarily changes the permissions of the file to allow you to write it. Otherwise, saving the file will fail.

---

2 Typically view is just a link to vi. Some systems arrange things such that view executes vim -R.

If you have a problem writing out the file, see the problem checklist in the section "Problems Saving Files" on page 13.

Recovering a Buffer

Occasionally a system failure may happen while you are editing a file. Ordinarily, any edits made after your last write (save) are lost. However, there is an option, -r, which lets you recover the edited buffer as it was at the time of a system crash.

Recovery in vi

On a traditional Unix system with the original vi, when you first log on after the system is running again, you will receive a mail message stating that your buffer has been saved. In addition, if you type the command:

    $ ex -r

or:

    $ vi -r

you will get a list of any files that the system has saved.

Use the -r option with a filename to recover the edited buffer. For example, to recover the edited buffer of the file *practice* after a system crash, enter:

    $ vi -r practice

It is wise to recover the file immediately, lest you inadvertently make edits to the file and then have to resolve a version skew between the preserved buffer and the newly edited file.

You can force the system to preserve your buffer even when there is not a crash by using the command :pre (short for :preserve). You may find it useful if you have made edits to a file and then discover that you can't save your edits because you don't have write permission. (You could also just write out a copy of the file under another name or into a directory where you do have write permission. See the section "Problems Saving Files" on page 13.)

Recovery in Vim

Recovery in Vim works somewhat differently. Vim usually keeps its working file (called a *swap file*) in the same directory as the file being edited. For *practice*, Vim's working file would be named *.practice.swp*.

If that file exists when you next go to edit *practice*, Vim asks you if you want to recover. You should do so, and write the file back out. You should then *quit immediately* and manually remove *.practice.swp*; Vim does not do that for you. After doing so, you may go back into Vim and continue editing your file normally.

The `directory` option to the `:set` command lets you control where Vim places the swap file. For more information, see the entry for `directory` in Table B-2 in the section "Vim 8.2 Options" on page 464.

Making Use of Registers

You have seen that while you are editing, your last deletion (`d` or `x`) or yank (`y`) is saved in the unnamed register. You can access the contents of that register and put the saved text back in your file with the put command (`p` or `P`).

The last nine deletions are stored in numbered registers. You can access any of these numbered registers to restore any (or all) of the last nine deletions. (Small deletions, of only parts of lines, are not saved in numbered registers, however. These deletions can be recovered only by using the `p` or `P` command immediately after you've made the deletion.)

You may also place yanks (copied text) into registers identified by letters. You can fill up to 26 registers (named a–z) with yanked text and restore that text with a put command at any time in your editing session.

Recovering Deletions

Being able to delete large blocks of text in a single bound is all very well and good, but what if you mistakenly delete 53 lines that you need? You can recover any of your past *nine* deletions, for they are saved in numbered registers. The last delete is saved in register 1, the second-to-last in register 2, and so on.

To recover a deletion, type `"` (double quote), identify the deleted text by number, and then give the put command. To recover your second-to-last deletion (from register 2), type:

> `"2p`

The deletion in register 2 is placed after the cursor.

If you're not sure which register contains the deletion you want to restore, you don't have to keep typing `"np` over and over again. You can use `"1p` to place the first delete text; if that's not right, use `u` to undo it. You can then use the repeat command (`.`) to place the next one, `u` to undo it, and so on. When you do this, the editor automatically increments the register number. As a result, you can search through the numbered registers using:

> `"1pu.u.u`   *etc.*

to put the contents of each succeeding register in the file one after the other. Each time you type `u`, the restored text is removed; when you type a dot (.), the contents of

the *next* register are restored to your file. Keep typing u and . until you've recovered the text you're looking for.

Yanking to Named Registers

You have seen that you must put (p or P) the contents of the unnamed register before you make any other edit, or else the register is overwritten. You can also use y and d with a set of 26 named registers (a–z) that are specifically available for copying and moving text. If you name a register to store the yanked text, you can retrieve the contents of the named register at any time during your editing session.

To yank into a named register, precede the yank command with a double quote (") and the character for the name of the register you want to load. For example:

```
"dyy          Yank the current line into register d
"a7yy         Yank the next seven lines into register a
```

After loading the named registers and moving to the new position, use p or P to put the text back:

```
"dP           Put the contents of register d before the cursor
"ap           Put the contents of register a after the cursor
```

There is no way to put part of a register into the text—it is all or nothing.

In the next chapter, you'll learn how to edit multiple files. Once you know how to travel between files without leaving the editor, you can use named registers to selectively transfer text between files. When using Vim's multiple-window feature, you can also use the unnamed deletion register to transfer data between files.

> The unnamed and named deletion registers are shared within the same Vim session, so you can easily copy/paste text between files being edited in multiple windows in one Vim session. But these buffers are *not* shared between multiple Vim sessions! (You might have gvim open on several files at once, for example.) However, gvim can access the system clipboard just like any other graphical application. So you can use GUI-level copy and paste to move text between files with no problems.

You can also delete text into named registers using much the same procedure:

```
"a5dd         Delete five lines into register a
```

If you specify a register name with a capital letter, your yanked or deleted text is *appended* to the current contents of the corresponding lowercase register. This allows you to be selective in what you move or copy. For example:

`"zd)`
>Delete from the cursor to the end of the current sentence and save the text in register `z`.

`2)`
>Move two sentences further on.

`"Zy)`
>Add the next sentence to register `z`. You can continue adding more text to a named register for as long as you like, but be warned: if you forget once, and you yank or delete to the register without specifying its name in capitalized form, you'll overwrite the register, losing whatever you had accumulated in it.

Marking Your Place

During an editing session, you can mark your place in the file with an invisible "bookmark," perform edits elsewhere, and then return to your marked place. Why would you need to do this? Will Gallego explains:

>One of my favorite uses of marking is deleting/yanking/modifying a large chunk of text. For example, say I want to delete a large number of lines. I might not want to count all those lines and then do *num*dd, but instead jump to the bottom, mark it with something like `ma` (mark, then `a` to use register `a` as the location), then jump to where I want to start deleting and hit `` d`a `` to delete the current line and all lines up to and including where `a` is. `yy` and other related commands may be used in a similar fashion.

Here's how you mark locations in command mode:

`m` *x*
>Mark the current position with *x* (*x* can be any letter). (The original `vi` allows only lowercase letters. Vim distinguishes between uppercase and lowercase letters.)

`'` *x (apostrophe)*
>Move the cursor to the first character of the line marked by *x*.

`` ` `` *x (backquote)*
>Move the cursor to the character marked by *x*.

`` `` `` *(backquotes)*
>Return to the exact position of the previous mark or context after a move.

`''` *(apostrophes)*
>Return to the beginning of the line of the previous mark or context.

Place markers are set only during the current session; they are not stored in the file.

Other Advanced Edits

There are other advanced edits that you can execute with vi and Vim, but to use them you must first learn a bit more about the ex editor by reading the next chapter.

Review of Register and Marking Commands

Table 4-2 summarizes the command-line options common to all versions of vi. Tables 4-3 and 4-4 summarize the register and marking commands.

Table 4-2. Command-line options

Option	Meaning
-c *n file*	Open *file* at line number *n* (POSIX standard version).
+*n file*	Open *file* at line number *n* (traditional vi version).
+ *file*	Open *file* at last line.
-c /*pattern file*	Open *file* at first occurrence of *pattern* (POSIX standard version)
+/*pattern file*	Open *file* at first occurrence of *pattern* (traditional vi version).
-c *command file*	Run *command* after opening *file*; usually a line number or search.
-r	Recover files after a crash.
-R	Operate in read-only mode (same as using view).

Table 4-3. Register names

Register names	Register use
1–9	The last nine deletions, from most to least recent.
a–z	Named registers for you to use as needed. Uppercase letters append to the register.

Table 4-4. Register and marking commands

Command	Meaning
"*bcommand*	Do *command* with register *b*.
m*x*	Mark current position with *x*.
'*x*	Move cursor to the first character of the line marked by *x*.
`*x*	Move cursor to the character marked by *x*.
``	Return to the exact position of the previous mark or context.
''	Return to the beginning of the line of the previous mark or context.

CHAPTER 5

Introducing the ex Editor

If this is a book on `vi` and Vim, why would we include a chapter on another editor? Well, `ex` is not really another editor. `vi` is the visual mode of the more general, underlying line editor, which is `ex`. Some `ex` commands can be useful to you while you are working in `vi`, since they can save you a lot of editing time. Most of these commands can be used without ever leaving `vi`: You can think of the `ex` command line as a third mode, alongside the regular command and insert modes.

The various `vi` motion and text-modification commands we've seen in the previous chapters are nice, but if that's all you've got, you may as well use Notepad or something similar. The reason `vi` lovers love `vi` is because of `ex`: `ex` *is where the power is!*

> Vim provides the underlying `ex` editor, with many enhancements over the original one. On systems where `vi` is Vim, `ex` usually also invokes Vim in `ex` mode.
>
> In this and the following chapters in Part I, we don't distinguish much between `vi` and Vim, since everything in these chapters applies to both. While reading, feel free to think of "`vi`" as standing for "`vi` and Vim."

You already know how to think of files as a sequence of numbered lines. `ex` gives you editing commands with greater mobility and scope. With `ex`, you can move easily between files and transfer text from one file to another in a variety of ways. You can quickly edit blocks of text larger than a single screen. And with global replacement you can make substitutions throughout a file for a given pattern.

This chapter introduces ex and its commands. You will learn how to:

- Move around a file by using line numbers
- Use ex commands to copy, move, and delete blocks of text
- Save files and parts of files
- Work with multiple files (reading in text or commands, traveling between files)

ex Commands

Long before vi or any other screen editor was invented, people communicated with computers on printing terminals, rather than on today's bitmapped screens with pointing devices and terminal emulation programs. Line numbers were a way to quickly identify a part of a file to be worked on, and line editors evolved to edit those files. A programmer or other computer user would typically print out a line (or lines) on the printing terminal, give the editing commands to change just that line, and then reprint to check the edited line.[1]

We don't edit files on printing terminals anymore, but some ex line editor commands are still useful to users of the more sophisticated visual editor built on top of ex. Although it is simpler to make many edits with vi, the line orientation of ex gives it an advantage when you want to make large-scale changes to more than one part of a file.

Many of the commands we'll see in this chapter have filename arguments. Although it's possible, it is usually a very bad idea to have spaces in your files' names. ex will be confused to no end, and you will go to more trouble than it's worth trying to get the filenames to be accepted. Use underscores, dashes, or periods to separate the components of your filenames, and you'll be much happier.

Before you start off simply memorizing ex commands (or worse, ignoring them), let's first take some of the mystery out of line editors. Seeing how ex works when it is invoked directly will help make sense of the sometimes obscure command syntax.

Open a file that is familiar to you and try a few ex commands. Just as you can invoke the vi editor on a file, you can invoke the ex line editor on a file by running the ex

---

1 ex is descended from the venerable Unix line editor ed, itself based on an earlier line editor known as QED. Versions of these editors for modern systems are still available.

command at the shell prompt. If you invoke ex, you will see a message about the total number of lines in the file, and a colon command prompt. For example:

```
$ ex practice
"practice" 8L, 261B
Entering Ex mode.  Type "visual" to go to Normal mode.
:
```

You won't see any lines in the file unless you give an ex command that causes one or more lines to be displayed.

ex commands consist of a line address (which can simply be a line number) plus a command; they are finished by hitting ENTER. One of the most basic commands is p for print (to the screen). So, for example, if you type 1p at the prompt, you see the first line of the file:

```
:1p
With a screen editor you can
:
```

In fact, you can leave off the p, because a line number by itself is equivalent to a print command for that line. To print more than one line, you can specify a range of line numbers (such as 1,3—two numbers separated by a comma, with or without spaces in between). For example:

```
:1,3
With a screen editor you can
scroll the page, move the cursor,
delete lines, insert characters, and more,
```

A command without a line number is assumed to affect the current line. So, for example, the substitute command (s), which allows you to substitute one word for another, could be entered like this:

```
:1
With a screen editor you can
:s/screen/line/
With a line editor you can
```

Notice that the changed line is reprinted after the command is issued. You could also make the same change like this:

```
:1s/screen/line/
With a line editor you can
```

Even though you will be invoking ex commands from vi and will not be using them directly, it is worthwhile to spend a few minutes in ex itself. You will get a feel for how you need to tell the editor which line (or lines) to work on, as well as which command to execute.

After you have given a few ex commands in your *practice* file, you should invoke vi on that same file, so that you can see it in the more familiar visual mode. When working in ex, the command :vi gets you from ex to vi.

To invoke an `ex` command from `vi`, you must type the special bottom-line character `:` (colon). Then type the `ex` command and press ENTER to execute it. So, for example, in the `ex` editor you move to a line simply by typing the number of the line at the colon prompt. To move to line 6 of a file using this command from within `vi`, enter:

    `:6`

And then press ENTER.

After the following exercise, we discuss `ex` commands only as they are executed from `vi`.

Exercise: The ex Editor

This exercise should be run from inside a terminal emulator window:

At the shell prompt, invoke the `ex` editor on a file called *practice*:	`ex practice`

A message appears:	`"practice" 8L, 261B`
	`Entering Ex mode.  Type "visual" to`
	`go to Normal mode.`

Go to and print (display) the first line:	`:1`
Print (display) lines 1 through 3:	`:1,3`
Substitute *line* for *screen* on line 1:	`:1s/screen/line/`
Invoke the `vi` editor on file:	`:vi`
Go to the first line:	`:1`

Problem Getting to Visual Mode

- *While editing in* `vi`, *you accidentally end up in the* `ex` *editor.*

  A `Q` in the command mode of `vi` invokes `ex`. Any time you are in `ex`, the command `vi` returns you to the `vi` editor.

Editing with ex

Many `ex` commands that perform normal editing operations have an equivalent in `vi` that does the job more simply. Obviously, you will use `dw` or `dd` to delete a single word or line rather than using the `delete` command in `ex`. However, when you want to make changes that affect numerous lines, you will find the `ex` commands more useful. They allow you to modify large blocks of text with a single command.

These `ex` commands are listed here, along with abbreviations for those commands. Remember that in `vi`, each `ex` command must be preceded with a colon. You can use the full command name or the abbreviation, whichever is easier to remember:

Full name	Abbreviation	Meaning
delete	d	Delete lines
move	m	Move lines
copy	co	Copy lines
	t	Copy lines (a synonym for co; short for "to")

You can separate the different elements of an ex command with spaces, if you find the command easier to read that way. For example, you can separate line addresses, patterns, and commands in this way. You cannot, however, use a space as a separator inside a pattern or at the end of a substitute command. (We show some examples later, in the section "Combining ex Commands" on page 74.)

Line Addresses

For each ex editing command, you have to tell ex which line number(s) to edit. And for the ex move and copy commands, you also need to tell ex where to move or copy the text to.

You can specify line addresses in several ways:

- With explicit line numbers
- With symbols that help you specify line numbers relative to your current position in the file
- With search patterns as *addresses* that identify the lines to be affected

Let's look at some examples.

Defining a Range of Lines

You can use line numbers to explicitly define a line or range of lines. Addresses that use explicit numbers are called *absolute* line addresses. For example:

:3,18d
    Delete lines 3 through 18, inclusive.

:160,224m23
    Move lines 160 through 224 to follow line 23. (Like delete and put in vi.)

:23,29co100
    Copy lines 23 through 29 and put them after line 100. (Like yank and put in vi.)

To make editing with line numbers easier, you can also display all line numbers on the left of the screen. The command:

    :set number

or its abbreviation

```
:set nu
```

displays line numbers. The file *practice* then appears:

```
1  With a line editor              "screen" changed to "line" earlier
2  you can scroll the page,
3  move the cursor, delete lines,
4  insert characters, and more
```

The displayed line numbers are not saved when you write a file, and they do not print if you print the file. Remember that long lines wrap on the screen but still count as a single line with respect to their numbers. Line numbers are displayed either until you quit the editing session or until you disable the set option:

```
:set nonumber
```

or:

```
:set nonu
```

Vim lets you toggle the setting with:

```
:set nu!
```

To temporarily display the line numbers for a set of lines, you can use the # sign as a command. For example:

```
:1,10#
```

displays the line numbers from line 1 to line 10.

As described in the section "Movement by Line Number" on page 51, you can also use the CTRL-G command to display the current line number. You can thus identify the line numbers corresponding to the start and end of a block of text by moving to the start of the block, typing CTRL-G, and then moving to the end of the block and typing CTRL-G again.

Yet another way to identify line numbers is with the = command in ex:

:=
    Print the total number of lines.

:.=
    Print the line number of the current line. (The period is a shorthand that means "the current line"; we discuss this next.)

:/*pattern*/=
    Print the line number of the next line that matches *pattern*. (The search starts from the current line. The use of search patterns is described shortly, in the section "Search Patterns" on page 72.)

Line-Addressing Symbols

Besides line numbers, you can also use symbols for line addresses. A dot (`.`) stands for the current line, and `$` stands for the last line of the file. `%` stands for every line in the file; it's the same as the combination `1,$`. These symbols can also be combined with absolute line addresses. For example:

`:.,$d`
> Delete from the current line to the end of the file.

`:20,.m$`
> Move from line 20 through the current line (inclusive) to the end of the file.

`:%d`
> Delete all the lines in the file.

`:%t$`
> Copy all the lines and place them at the end of the file (making a consecutive duplicate).

In addition to an absolute line address, you can specify an address relative to the current line. The symbols + and - work like arithmetic operators. When placed before a number, these symbols add or subtract the value that follows. For example:

`:.,.+20d`
> Delete from the current line through the next 20 lines.

`:226,$m.-2`
> Move lines 226 through the end of the file to two lines above the current line.

`:.,+20#`
> Display line numbers from the current line to 20 lines further on in the file.

In fact, you don't need to type the dot (.) when you use + or - because the current line is the assumed starting position.

Without a number following them, + and - are equivalent to +1 and –1, respectively.[2] Similarly, ++ and -- each extend the range by an additional line, and so on. Thus `:+++` moves you forward by three lines. The + and - can also be used with search patterns, as shown in the next section.

---

2 In a relative address, you shouldn't separate the plus or minus symbol from the number that follows it. For example, `+10` means "10 lines following," but `+ 10` means "11 lines following (`1 + 10`)," which is probably not what you mean (or want).

The number `0` stands for the top of the file (imaginary line zero). `0` is equivalent to `1-`, and both allow you to move or copy lines to the very start of a file, before the first line of existing text. For example:

`:-,+t0`
Copy three lines (the line above the cursor through the line below the cursor) and put them at the top of the file.

Search Patterns

Another way that `ex` can address lines is by using search patterns. For example:

`:/pattern/d`
Delete the next line containing *pattern*.

`:/pattern/+d`
Delete the line *below* the next line containing *pattern*. (You could also use `+1` instead of `+` alone.)

`:/pattern1/,/pattern2/d`
Delete from the first line containing *pattern1* through the first line containing *pattern2*.

`:.,/pattern/m23`
Take the text from the current line (`.`) through the first line containing *pattern* and put it after line 23.

Note that a pattern is delimited by a slash both *before* and *after*. Any spaces or tab characters between the slashes are interpreted as part of the pattern to search for.

Use `?` as the delimiter instead of `/` if you want to search backward in the file.

If you make deletions by pattern with `vi` and `ex`, there is a difference in the way the two editors operate. Suppose your file *practice* contains the lines:

```
With a screen editor you can scroll the
page, move the cursor, delete lines, insert
characters, and more, while seeing results
of your edits as you make them.
```

To delete through the word *while*, do the following:

Keystrokes	Results

`d/while`	`With a screen editor you can scroll the`
	`page, move the cursor, while seeing results`
	`of your edits as you make them.`

	The `vi` delete to *pattern* command deletes from the cursor up to the word *while* but leaves the remainder of both lines.

Keystrokes	Results
`:.,/while/d`	With a screen editor you can scroll the █f your edits as you make them.
	The `ex` command deletes the entire range of addressed lines—in this case, both the current line and the line containing the pattern. All lines are deleted in their entirety.

Redefining the Current Line Position

Sometimes, using a relative line address in a command can give you unexpected results. For example, suppose the cursor is on line 1 and you want to print line 100 plus the five lines below it. If you type:

`:100,+5 p`

you'll get an error message from Vim saying, "E16: Invalid range." `vi` will tell you, "First address exceeds second." The reason the command fails is that the second address is calculated relative to the current cursor position (line 1), so your command is really saying this:

`:100,6 p`

What you need is some way to tell the command to think of line 100 as the "current line," even though the cursor is on line 1.

`ex` provides such a way. When you use a semicolon instead of a comma, the first line address is recalculated as the current line. For example, the command:

`:100;+5 p`

prints the desired lines. The +5 is now calculated relative to line 100. A semicolon is useful with search patterns as well as with absolute addresses. For example, to print the next line containing *pattern*, plus the 10 lines that follow it, enter the command:

`:/pattern/;+10 p`

Global Searches

You already know how to use `/` (slash) to search for patterns of characters in your files. `ex` has a global command, `g`, that lets you search for a pattern and display all lines containing the pattern when it finds them. The command `:g!` does the opposite of `:g`. Use `:g!` (or its synonym, `:v`) to search for all lines that do *not* contain *pattern*.

You can use the global command on all lines in the file, or you can use line addresses to limit a global search to specified lines or to a range of lines:

`:g/pattern`
    Find (move to) the last occurrence of *pattern* in the file.

`:g/`*pattern*`/p`
> Find and display all lines in the file containing *pattern*. Vim displays the line and then prompts you, "Press ENTER or type command to continue."

`:g!/`*pattern*`/nu`
> Find and display all lines in the file that don't contain *pattern*; also display the line number for each line found.

`:60,124g/`*pattern*`/p`
> Find and display any lines between lines 60 and 124 containing *pattern*.

As you might expect, `g` can also be used for global replacements. We'll talk about that in Chapter 6, "Global Replacement".

Combining ex Commands

You don't always need to type a colon to begin a new `ex` command. In `ex`, the vertical bar (`|`) is a command separator, allowing you to combine multiple commands from the same `ex` prompt (in much the same way that a semicolon separates multiple commands at the shell prompt). When you use the `|`, keep track of the line addresses you specify. If one command affects the order of lines in the file, the next command does its work using the new line positions. For example:

`:1,3d | s/thier/their/`
> Delete lines 1 through 3 (leaving you now on the top line of the file), and then make a substitution on the current line (which was line 4 before you invoked the `ex` prompt).

`:1,5 m 10 | g/`*pattern*`/nu`
> Move lines 1 through 5 after line 10, and then display all lines (with numbers) containing *pattern*.

Note the use of spaces to make the commands easier to read.

Saving and Exiting Files

> **I Am Devloper**
> @iamdevloper
>
> I've been using Vim for about 2 years now, mostly because I can't figure out how to exit it.
>
> 1:26 AM · Feb 18, 2014 · Tweetbot for iOS

Figure 5-1. Not everyone gets `vi` *(from* https://twitter.com/iamdevloper/status/435555976687923200, *used with permission)*

Unlike IAmDevloper (see Figure 5-1), you have learned the `vi` command `ZZ` to write (save) your file and quit. But you will frequently want to exit a file using `ex` commands, because these commands give you greater control. We've already mentioned some of these commands in passing. Now let's take a more formal look:

`:w`

> Write (save) the buffer to the file but do not exit. You can (and should) use `:w` throughout your editing session to protect your edits against system failure or a major editing error.

`:q`

> Quit the editor (and return to the shell prompt).

`:wq`

> Write the file and then quit the editor. The write happens unconditionally, even if the file was not changed. This updates the modification time of the file.

`:x`

> Write the file and then quit (e**x**it) the editor. The file is written only if it has been modified.[3]

The editor protects existing files and your edits in the buffer. For example, if you want to write your buffer to an existing file, you will get a warning. Likewise, if you have invoked `vi` on a file, made edits, and want to quit *without* saving the edits, `vi` gives you an error message, such as:

```
No write since last change.
```

These warnings can prevent costly mistakes, but sometimes you want to proceed with the command anyway. An exclamation point (`!`) after your command overrides the warning:

```
:w!
:q!
```

Vim helpfully tells you this when you try to quit without saving the file:

```
E37: No write since last change (add ! to override)
```

`:w!` can also be used to save edits in a file that was opened in read-only mode with `vi -R` or `view` (assuming you have write permission for the file).

`:q!` is an essential editing command that allows you to quit without affecting the original file, regardless of any changes you made in the session.[4] The contents of the buffer are discarded.

---

3 The difference between `:wq` and `:x` is important when editing source code and using `make`, which performs actions based on file modification times.

Renaming the Buffer

You can also use `:w` to save the entire buffer (the copy of the file you are editing) under a new filename.

Suppose you have a file *practice*, which contains 600 lines. You open the file and make extensive edits. You want to quit but also save *both* the old version of *practice* and your new edits for comparison. To save the edited buffer in a file called *practice.new*, give the command:

    :w practice.new

Your old version, in the file *practice*, remains unchanged (provided that you didn't previously use `:w`). You can now quit editing the new version by typing `:q`.

Saving Part of a File

While editing, you will sometimes want to save just part of your file as a separate, new file. For example, you might have entered formatting codes and text that you want to use as a header for several files.

You can combine `ex` line addressing with the write command, `w`, to save part of a file. For example, if you are in the file *practice* and want to save part of *practice* as the file *newfile*, you could enter:

`:230,$w newfile`
> Save from line 230 to end of file in *newfile*.

`:.,600w newfile`
> Save from the current line to line 600 in *newfile*.

Appending to a Saved File

You can use the Unix redirect and append operator (`>>`) with `w` to append all or part of the contents of the buffer to an existing file. For example, if you entered:

    :1,10w newfile

and then:

    :340,$w >> newfile

newfile would contain lines 1–10 and from line 340 to the end of the buffer.

---

4 Only changes made since the last write are thrown away.

Copying a File into Another File

Sometimes you want to copy text or data from some other existing file into the file you are editing. You read in the contents of another file with the `ex` command:

    :read *filename*

or its abbreviation:

    :r *filename*

This command inserts the contents of *filename* starting on the line after the cursor position in the file. If you want to specify a line other than the one the cursor's on, simply type the line number (or other line address) you want before the `:read` or `:r` command.

Let's suppose you are editing the file *practice* and want to read in a file called *data* from another directory called */home/tim*. Position the cursor one line above the line where you want the new data inserted, and enter:

    :r /home/tim/data

The entire contents of */home/tim/data* are read into *practice*, beginning below the line with the cursor.

To read in the same file and place it after line 185, you would enter:

    :185r /home/tim/data

Here are other ways to read in a file:

`:$r /home/tim/data`
: Place the read-in file at the end of the current file.

`:0r /home/tim/data`
: Place the read-in file at the very beginning of the current file.

`:/pattern/r /home/tim/data`
: Place the read-in file in the current file, after the line containing *pattern*.

Editing Multiple Files

`ex` commands enable you to switch between multiple files. The advantage of editing multiple files is speed; it takes time to exit and reenter `vi` or Vim for each file you want to edit. Staying in the same editing session and traveling between files is not only faster for access, but you also save abbreviations and command sequences that you have defined (see Chapter 7, "Advanced Editing"), and you keep yank registers so that you can copy text from one file to another.

Invoking Vim on Multiple Files

When you first invoke the editor, you can name more than one file to edit and then
use ex commands to travel between the files. For example:

```
$ vim file1 file2
```

edits *file1* first. After you have finished editing the first file, the ex command `:w` writes
(saves) *file1* and `:n` calls in the next file (*file2*).

Suppose you want to edit two files, *practice* and *note*:

Keystrokes	Results
$ `vim practice note`	▌With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more,
	Open the two files *practice* and *note*. The first-named file, *practice*, appears on your screen. Perform any edits.
`:w`	"practice" 8L, 261C 8,1 All
	Save the edited file *practice* with the ex command w.
`:n`	▌Dear Mr. Henshaw: Thank you for the prompt . . .
	Call in the next file, *note*, with the ex command n. Perform any edits.
`:x`	"note" 19L, 571C written 19,1 All
	Save the second file, *note*, and quit the editing session.

Using the Argument List

ex actually lets you do more than just move to the next file in the argument list
with `:n`. The `:args` command (abbreviated `:ar`) lists the files named on the com-
mand line, with the current file enclosed in brackets:

Keystrokes	Results
$ `vim practice note`	▌With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more,
	Open the two files *practice* and *note*. The first-named file, *practice*, appears on your screen.
`:args`	[practice] note 8,1 All
	Vim displays the argument list in the status line, with brackets around the current filename.

vi's `:rewind` (`:rew`) command resets the current file to be the first file named on the
command line. Vim provides a corresponding `:last` command to move to the last
file on the command line. If you just want to go back to the previous file, Vim gives
you the `:prev` command.

Calling in New Files

You don't have to call in multiple files at the beginning of your editing session. You can switch to another file at any time with the ex command `:e`. If you want to edit another file, you first need to save your current file (`:w`) and then give the command:

    :e filename

Suppose you are editing the file *practice* and want to edit the file *letter* and then return to *practice*:

Keystrokes	Results
`:w`	`"practice" 8L, 261C` `8,1` `All`
	Save *practice* with `:w`. *practice* is saved and remains on the screen. You can now switch to another file, because your edits are saved.
`:e letter`	`"letter" 23L, 1344C` `1,1` `All`
	Call in the file *letter* with `:e`. Perform any edits.

Filename Shortcuts

The editor "remembers" two filenames at a time as the current and alternate filenames. These can be referred to by the symbols `%` (current filename) and `#` (alternate filename). `#` is particularly useful with `:e`, since it allows you to switch easily back and forth between two files. In the example just given, you could return to the first file, *practice*, by typing the command `:e #`. You could also read the file *practice* into the current file by typing `:r #`.

If you have not first saved the current file, the editor does not allow you to switch files with `:e` or `:n` unless you tell it imperatively to do so by adding an exclamation point after the command.

For example, if after making some edits to *letter*, you wanted to discard the edits and return to *practice*, you could type `:e! #`.

The following command is also useful; it discards your edits and returns to the last saved version of the current file:

    :e!

In contrast to the `#` symbol, `%` is useful mainly when writing out the contents of the current buffer to a new file. For example, in the earlier section "Renaming the Buffer" on page 76, we showed you how to save a second version of the file *practice* with the command:

    :w practice.new

Since `%` stands for the current filename, that line could also have been typed:

```
:w %.new
```

Switching Files from Command Mode

Since switching back to the previous file is something that you will tend to do a lot, you don't have to move to the ex command line to do it. The vi command CTRL-^ (the CTRL key plus the caret key) does this for you. Using this command is the same as typing `:e #`. As with the `:e` command, if the current buffer has not been saved, the editor does not let you switch back to the previous file.

Edits Between Files

When you give a yank register a one-letter name, you have a convenient way to move text from one file to another. Named registers are not cleared when a new file is loaded into the editing buffer with the `:e` command. Thus, by yanking or deleting text from one file (into multiple named registers if necessary), calling in a new file with `:e`, and putting the named register(s) into the new file, you can transfer material between files.

The following example illustrates how to transfer text from one file to another:

Keystrokes	Results
`"f4yy`	With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of the edits as you make them
	Yank four lines into register f.
`:w`	"practice" 8L, 261C 8,1 All
	Save the file.
`:e letter`	Dear Mr. Henshaw: I thought that you would be interested to know that: Yours truly,
	Enter the file *letter* with `:e`. Move the cursor to where the copied text will be placed.
`"fp`	Dear Mr. Henshaw: I thought that you would be interested to know that: With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of the edits as you make them Yours truly,
	Place yanked text from named register f below the cursor.

Another way to move text from one file to another is to use the ex commands `:ya` (yank) and `:pu` (put). These commands work the same way as the equivalent vi commands y and p, but they are used with ex's line-addressing capability and named registers.

For example:

```
:160,224ya  a
```

yanks (copies) lines 160 through 224 into register a. Next, you would move with `:e` to the file where you want to put these lines. Place the cursor on the line where you want to put the yanked lines. Then type:

```
:pu a
```

to put the contents of register a after the current line.

ex Command Summaries

Here are summary tables of the ex commands presented in this chapter (see Tables 5-1 through 5-7). Appendix A, "The vi, ex, and Vim Editors", provides a fuller command reference for the most useful ex commands in vi and Vim.

Table 5-1. Line printing commands

Full name	Abbreviation	Meaning
Address		Print line at *address*
Address range		Print lines at *address range*
print	p	Print lines
	#	Print lines with line numbers

Table 5-2. Line deletion, moving, and copying

Full name	Abbreviation	Meaning
delete	d	Delete lines
move	m	Move lines
copy	co	Copy lines
	t	Copy lines (a synonym for co; short for "to")
yank	ya	Yank lines into a named register
put	pu	Put lines from a named register

Table 5-3. Line-addressing symbols

Symbol	Meaning
n	Line number *n*
.	Current line
$	Last line
%	All lines in the file
. +*n*	Current line plus *n*
. -*n*	Current line minus *n*
/*pattern*/	Search forward for first line that matches *pattern*
?*pattern*?	Search backward for first line that matches *pattern*

Table 5-4. Operating globally

Full name	Abbreviation	Meaning
global *command*	g *command*	Globally (on all lines) execute *command*
global! *pattern command*	g! *pattern command*	On all lines not matching *pattern* execute *command*
	v *pattern command*	On all lines not matching *pattern* execute *command*

Table 5-5. Working with buffers and files

Full name	Abbreviation	Meaning
args	ar	Show the argument list, with the current filename in square brackets
edit	e	Switch to editing the named file
last	la	Go to the last file in the argument list
next	n	Move on to the next file named on the command line
previous	prev	Go back to the previous file
read	r	Read the named file into the editing buffer
rewind	rew	Go back to the first file in the argument list
write	w	Write the editing buffer to disk
CTRL-^		Go back to the previous file (vi command)

Table 5-6. Exiting the editor

Full name	Abbreviation	Meaning
`quit`	`q`	Quit the editor
	`wq`	Write the file unconditionally, and then quit
`xit`	`x`	Write the file only if it has changed, and then quit
`Q`		Switch to `ex` (`vi` command)
`visual`	`vi`	Switch from `ex` to `vi`

Table 5-7. Filename shorthands

Character	Meaning
`%`	Current filename
`#`	Previous filename

Global Replacement

Sometimes, halfway through a document or at the end of a draft, you may recognize inconsistencies in the way that you refer to certain things. Or, in a manual, some product whose name appears throughout your file is suddenly renamed (marketing!). Often enough it happens that you have to go back and change what you've already written, and you need to make similar or identical changes in several places.

The way to make these changes is with a powerful feature called *global replacement*. With one command you can automatically replace a word (or a string of characters) wherever it occurs in the file.

In a global replacement, the `ex` editor checks each line of a file for a given pattern of characters. On all lines where the pattern is found, `ex` replaces the pattern with a *new string* of characters. For right now, we'll treat the search pattern as if it were a simple string; later in the chapter we'll look at the powerful pattern-matching language known as *regular expressions*.

Global replacement really uses two `ex` commands: `:g` (global) and `:s` (substitute). Since the syntax of global replacement commands can get fairly complex, let's look at it in stages.

The Substitute Command

The substitute command has the syntax:

```
:s/old/new/
```

This changes the *first* occurrence of the pattern *old* to *new* on the current line. The `/` (slash) is the delimiter between the various parts of the command and is optional when it is the last character on the line. (Actually, you can use any punctuation character for the delimiter; this is discussed later in the chapter.)

The `:s` command allows options following the substitution string. For example, a substitute command with the syntax:

    :s/*old*/*new*/g

changes *every* occurrence of *old* to *new* on the current line, not just the first occurrence. The `g` option in this syntax also stands for *global*. (The `g` option affects each pattern on a line; don't confuse it with the `:g` command, discussed shortly, which affects each line of a file.)

By prefixing the `:s` command with addresses, you can extend its range to more than one line. For example, this command changes every occurrence of *old* to *new* from line 50 to line 100:

    :50,100s/*old*/*new*/g

The following command changes every occurrence of *old* to *new* within the entire file:

    :1,$s/*old*/*new*/g

You can also use `%` instead of `1,$` to specify every line in a file. Thus, the preceding command could also be given like this:

    :%s/*old*/*new*/g

Global replacement is much faster than finding each instance of a string and replacing it individually. Because the command can be used to make many different kinds of changes, and because it is so powerful, we first illustrate simple replacements and then build up to complex, context-sensitive replacements.

Confirming Substitutions

It makes sense to be overly careful when using a search and replace command. It sometimes happens that what you get is not what you expect. You can undo any search and replacement command by entering u, provided that the command was the most recent edit you made. But you don't always catch undesired changes until it is too late to undo them.

Another way to protect your edited file is to save the file with `:w` before performing a global replacement. Then at least you can quit the file without saving your edits and go back to where you were before the change was made. You can also read the last saved version of the file back in with `:e!`. (Saving your file is a good idea in any case.)

It's wise to be cautious and know exactly what is going to be changed in your file. If you'd like to see what the search turns up and confirm each replacement before it is made, add the `c` option (for "confirm") at the end of the substitute command:

    :1,30s/his/the/gc

ex (in Vim) displays the entire line where the string has been located, highlights the text to be replaced, and prompts for confirmation:

```
copyists at his school
~
~
~
replace with the (y/n/a/q/l/^E/^Y)?█
```

If you want to make the replacement, you must enter y (for yes). If you don't want to make a change, simply press n (for no).

Quoting from the Vim documentation, here is the meaning of the possible responses:

y	to substitute this match
n	to skip this match
a	to substitute this and all remaining matches
q	to quit substituting
l	to substitute this match and then quit ("last")
CTRL-E	to scroll the screen up
CTRL-Y	to scroll the screen down
ESC	to quit substituting

Vim provides a number of additional options beyond just g and c. Issue the command :help s_flags for more information.

The combination of the vi commands n (repeat last search) and dot (.) (repeat last command) is also an extraordinarily useful and quick way to page through a file and make repetitive changes that you may not want to make globally. So, for example, if your (human) editor has told you that you're using *which* when you should be using *that*, you can spot-check every occurrence of *which*, changing only those that are incorrect:

/which	Search for *which*
cwthat ESC	Change to *that*
n	Repeat search
n	Repeat search, skip a change
.	Repeat change (if appropriate)
	(Etc.)

Doing Things Globally Across the File

`ex` provides a powerful command that lets you apply a second command across all relevant lines in a file. This is the *global* command, `:g`, which has the form:

```
:g/pattern/ command
```

Upon receiving this command, `ex` traverses the entire editing buffer, remembering each line that matches *pattern*. Then, for each line that matched, it executes the given *command*. Here are two examples:

`g/# FIXME/d`
> Delete all lines with "FIXME" comments on them.

`g/# FIXME/s/FIXME/DONE/`
> Change all instances of "FIXME" comments to say "DONE."

The global command (`:g`) is most often used in tandem with the substitute command (`s`), as we are about to see. But you can combine it with other `ex` commands as well, which we describe later in the chapter.

Context-Sensitive Replacement

The simplest global replacements substitute one word or phrase for another. If you have typed a file with several misspellings (*editer* for *editor*), you can do the global replacement:

```
:%s/editer/editor/g
```

This substitutes *editor* for every occurrence of *editer* throughout the file.

Using the global command, `:g`, you can search for a pattern, and then, once you find the line with the pattern, make a substitution on a string different from the pattern. You can think of this as context-sensitive replacement.

The syntax is as follows:

```
:g/pattern/s/old/new/g
```

The first `g` tells the command to operate on all lines of the file that match *pattern*. On those lines containing *pattern*, `ex` is to replace (substitute, `s`) the characters matching *old* with the characters in *new*. The last `g` indicates that the substitution is to occur globally *on that line*. This means that all matches of *old* are replaced by *new*, not just the first match on each relevant line.

For example, as we write this book, the HTML directives `<span class="keycap">` and `</span>` embedded in the AsciiDoc place a box around ESC to show the Escape key. You want ESC to be all in caps, but you don't want to change any instances of

---

*Esc*ape that might be in the text. To change instances of *Esc* to *ESC* only when *Esc* is on a line that contains the `class="keycap"` notation, you could enter:

```
:g/class="keycap"/s/Esc/ESC/g
```

If the pattern being used to find the line is the same as the one you want to change, you don't have to repeat it. The command:

```
:g/string/s//new/g
```

searches for lines containing *string* and substitutes for that same *string*.

Note that:

```
:g/editer/s//editor/g
```

has the same effect as:

```
:%s/editer/editor/g
```

You can save some typing by using the second form. As we mentioned earlier, you can combine the `:g` command with `:d`, `:m`, `:co`, and other ex commands besides `:s`. As we'll show, you can thus make global deletions, moves, and copies.

Pattern-Matching Rules

In making global replacements, Unix editors such as `vi` and Vim allow you to search not just for fixed strings of characters but also for variable patterns of words, referred to as *regular expressions*.

When you specify a literal string of characters, the search might turn up other occurrences that you didn't want to match. The problem with searching for words in a file is that a word can be used in different ways, or one word might be embedded in another (consider the "top" in "stopper"). Regular expressions help you conduct a search for words in context. Note that regular expressions can be used with the search commands `/` and `?`, as well as in the ex commands `:g` and `:s`.

For the most part, the same regular expressions work with other Unix programs, such as `grep`, `sed`, and `awk`.[1]

Regular expressions are made up by combining normal characters with a number of special characters called *metacharacters*.[2] The metacharacters and their uses are listed next.

---

1 Much more information on regular expressions can be found in the two O'Reilly books *sed & awk*, 2nd ed. by Dale Dougherty and Arnold Robbins, and *Mastering Regular Expressions*, 3rd ed. by Jeffrey E. F. Friedl.

2 Technically speaking, we should probably call these *metasequences*, since sometimes two characters together, not just a single character, have special meaning. Nevertheless, the term *metacharacters* is in common use in Unix literature, so we follow that convention here.

Metacharacters Used in Search Patterns

Here are the metacharacters and what they do:

. *(period, dot)*

Match any *single* character except a newline. Remember that spaces are treated as characters. For example, `p.p` matches character strings such as *pep*, *pip*, and *pcp*.

*

Match zero or more (as many as there are) of the single character that immediately precedes it. For example, `slo*w` matches *slow* (one *o*) or *slw* (no *o*). (It also matches *sloow*, *slooow*, and so on.)

The `*` can follow a metacharacter. For example, since `.` (dot) means any character, `.*` means "match any number of any character."

Here's a specific example of this: the command `:s/End.*/End/` removes all characters after *End* (it replaces the remainder of the line with nothing).

^

When used at the start of a regular expression, require that the following regular expression be found at the beginning of the line. For example, `^Part` matches *Part* when it occurs at the beginning of a line, and `^...` matches the first three characters of a line. When not at the beginning of a regular expression, `^` stands for itself.

$

When used at the end of a regular expression, require that the preceding regular expression be found at the end of the line; for example, `here:$` matches only when *here:* occurs at the end of a line. When not at the end of a regular expression, `$` stands for itself.

The `^` and `$` are referred to as *anchors*, since they anchor the match to the beginning or end of the line, respectively.

****

Treat the following special character as an ordinary character. For example, `\.` matches an actual period instead of "any single character," and `\*` matches an actual asterisk instead of "any number of a character." The \ (backslash) prevents the interpretation of a special character. This prevention is called "escaping the character." Use `\\` to get a literal backslash.

[ ]

Match any *one* of the characters enclosed between the brackets. For example, `[AB]` matches either *A* or *B*, and `p[aeiou]t` matches *pat*, *pet*, *pit*, *pot*, or *put*. You specify a range of consecutive characters by separating the first and last

characters in the range with a hyphen. For example, [A-Z] matches any upper-case letter from *A* to *Z*, and [0-9] matches any digit from *0* to *9*.

You can include more than one range inside brackets, and you can specify a mix of ranges and separate characters. For example, [:;A-Za-z()] matches four different punctuation marks, plus all the English letters.

> When regular expressions and vi were first developed, they were meant to work only with the ASCII character set. In today's global market, modern systems support *locales*, which provide different interpretations of the characters that lie between a and z. To get accurate results, you should use POSIX bracket expressions (discussed shortly) in your regular expressions, and avoid ranges of the form a-z.

Most metacharacters lose their special meaning inside brackets, so you don't need to escape them if you want to use them as ordinary characters. Within brackets, the three metacharacters you still need to escape are \, -, and]. The hyphen (-) acquires meaning as a range specifier; to use an actual hyphen, you can also place it as the first character inside the brackets. (This is also true of].)

A caret (^) has special meaning only when it is the first character inside the brackets, but in this case the meaning differs from that of the normal ^ meta-character. As the first character within brackets, the ^ reverses their sense: the brackets match any one character *not* in the list. For example, [^0-9] matches any character that is not a digit.

\(\)

Save the subpattern enclosed between \(and \) into a special holding space, or a *hold buffer*.[3] Up to nine subpatterns can be saved in this way on a single line. For example, the pattern

```
\(That\) or \(this\)
```

saves *That* in hold buffer number 1 and saves *this* in hold buffer number 2. The text matching the held subpatterns can be "replayed" in substitutions by the sequences \1 to \9. For example, to rephrase *That or this* to read *this or That*, you could enter:

```
:%s/\(That\) or \(this\)/\2 or \1/
```

You can also use the *n* notation within a search or substitute string. For example:

```
:s/\(abcd\)\1/alphabet-soup/
```

---

3 The hold buffer is different from both the file editing buffer and the text deletion registers.

changes *abcdabcd* into *alphabet-soup*.

`\< \>`

Match characters at the beginning (`\<`) or at the end (`\>`) of a word. The end or beginning of a word is determined either by a punctuation mark or by a space. For example, the expression `\<ac` matches only words that begin with *ac*, such as *action*. The expression `ac\>` matches only words that end with *ac*, such as *maniac*. Neither expression matches *react*. Note that unlike `\(…\)`, these do not have to be used in matched pairs.

In the original `vi`, there is an additional metacharacter:

`~`

Match whatever regular expression was used in the *last* search. For example, if you searched for *The*, you could search for *Then* with `/~n`. Note that you can use this pattern only in a regular search (with `/`). It won't work as the pattern in a substitute command. It does, however, have a similar meaning in the replacement portion of a substitute command. (This is described shortly, in the section "Metacharacters Used in Replacement Strings" on page 94.)

This use of `~` is a rather flaky feature of the original `vi`. After using it, the saved search pattern is set to the *new* text typed after the `~`, *not* the combined new pattern, as one might expect. While this feature exists, it has little to recommend its use. Also, Vim does not behave this way.

Note that Vim supports an extended regular expression syntax. See the section "Extended Regular Expressions" on page 176 for more information.

POSIX Bracket Expressions

We have just described the use of brackets for matching any one of the enclosed characters, such as `[a-z]`. The POSIX standard introduced additional facilities for matching characters that are not in the English alphabet. For example, the French *è* is an alphabetic character, but the typical character class `[a-z]` would not match it. Additionally, the standard provides for sequences of characters that should be treated as a single unit when matching and collating (sorting) string data.

POSIX also formalizes the terminology. Groups of characters within brackets are called *bracket expressions* in the POSIX standard. Within bracket expressions, alongside literal characters such as *a*, *!*, and so on, you can have additional components. These components are:

Character classes

A POSIX character class consists of keywords bracketed by `[:` and `:]`. The keywords describe different classes of characters, such as alphabetic characters, control characters, and so on (see Table 6-1).

Collating symbols

A collating symbol is a multicharacter sequence that should be treated as a unit. It consists of the characters bracketed by `[.` and `.]`.

Equivalence classes

An equivalence class lists a set of characters that should be considered equivalent, such as *e* and *è*. It consists of a named element from the locale, bracketed by `[=` and `=]`.

All three of these constructs *must* appear inside the square brackets of a bracket expression. For example, `[[:alpha:]!]` matches any single alphabetic character or the exclamation point, `[[.ch.]]` matches the collating element *ch* but does not match just the letter *c* or the letter *h*. In a French locale, `[[=e=]]` might match any of *e*, *è*, or *é*. Classes and matching characters are shown in Table 6-1.

Table 6-1. POSIX character classes

Class	Matching characters
`[:alnum:]`	Alphanumeric characters
`[:alpha:]`	Alphabetic characters
`[:blank:]`	Space and tab characters only
`[:cntrl:]`	Control characters
`[:digit:]`	Numeric characters
`[:graph:]`	Printable and visible (nonspace) characters
`[:lower:]`	Lowercase characters
`[:print:]`	Printable characters (includes whitespace)
`[:punct:]`	Punctuation characters
`[:space:]`	All whitespace characters (space, tab, newline, vertical tab, etc.)
`[:upper:]`	Uppercase characters
`[:xdigit:]`	Hexadecimal digits

Modern systems are sensitive to the locale chosen at installation time; you can expect to get reasonable results, particularly when trying to match only lowercase or uppercase letters, just by using the POSIX bracket expressions.[4]

---

4 On Solaris 10, */usr/xpg4/bin/vi* and */usr/xpg6/bin/vi* support POSIX bracket expressions, but */usr/bin/vi* does not. On Solaris 11, all versions do support POSIX bracket expressions.

<div style="border:1px solid black; padding:10px;">

How Do I Choose My Locale?

You choose the locale for commands to use by setting certain environment variables, whose names begin with the characters `LC_`. The details are beyond the scope of this book, other than to say that the simplest way to set a locale is to set the `LC_ALL` environment variable. A default locale is set up when the system is installed, if you do not override it.

You can see the list of available locales on your system, usually with the `locale` command:

```
$ locale -a        On GNU/Linux
C
C.UTF-8
en_AG
en_AG.utf8
en_AU.utf8
...
```

Note that files *do not* have locales associated with them; locales specify how commands treat the data they read from files. Typically, files encoded in UTF-8 should be properly handled in all Unicode-based locales, but your mileage may vary.

</div>

Metacharacters Used in Replacement Strings

When you make global replacements, the regular expression metacharacters discussed earlier carry their special meanings only within the search portion (the first part) of the command.

For example, when you type this:

```
:%s/1\.  Start/2.  Next, start with $100/
```

note that the replacement string treats the characters `.` and `$` literally, without your having to escape them. By the same token, let's say you enter:

```
:%s/[ABC]/[abc]/g
```

If you're hoping to replace *A* with *a*, *B* with *b*, and *C* with *c*, you'll be surprised. Since brackets behave like ordinary characters in a replacement string, this command changes every occurrence of *A*, *B*, or *C* to the five-character string *[abc]*.

To solve problems like this, you need a way to specify variable replacement strings. Fortunately, there are additional metacharacters that have special meaning in a *replacement* string:

n

Replace the *n* with the text matching the *n*th subpattern previously saved by \(and \), where *n* is a number from 1 to 9, and previously saved subpatterns

<div style="border:1px solid black; padding:10px;">

How Do I Choose My Locale?

You choose the locale for commands to use by setting certain environment variables, whose names begin with the characters `LC_`. The details are beyond the scope of this book, other than to say that the simplest way to set a locale is to set the `LC_ALL` environment variable. A default locale is set up when the system is installed, if you do not override it.

You can see the list of available locales on your system, usually with the `locale` command:

```
$ locale -a        On GNU/Linux
C
C.UTF-8
en_AG
en_AG.utf8
en_AU.utf8
...
```

Note that files *do not* have locales associated with them; locales specify how commands treat the data they read from files. Typically, files encoded in UTF-8 should be properly handled in all Unicode-based locales, but your mileage may vary.

</div>

Metacharacters Used in Replacement Strings

When you make global replacements, the regular expression metacharacters discussed earlier carry their special meanings only within the search portion (the first part) of the command.

For example, when you type this:

```
:%s/1\.  Start/2.  Next, start with $100/
```

note that the replacement string treats the characters `.` and `$` literally, without your having to escape them. By the same token, let's say you enter:

```
:%s/[ABC]/[abc]/g
```

If you're hoping to replace *A* with *a*, *B* with *b*, and *C* with *c*, you'll be surprised. Since brackets behave like ordinary characters in a replacement string, this command changes every occurrence of *A*, *B*, or *C* to the five-character string *[abc]*.

To solve problems like this, you need a way to specify variable replacement strings. Fortunately, there are additional metacharacters that have special meaning in a *replacement* string:

n

Replace the *n* with the text matching the *n*th subpattern previously saved by \(and \), where *n* is a number from 1 to 9, and previously saved subpatterns

(kept in hold buffers) are counted from the left on the line. See the explanation for \(and \) in the earlier section

\

Treat the following special character as an ordinary character. Backslashes are metacharacters in replacement strings as well as in search patterns. To specify a real backslash, type two in a row (\\).

&

Replace the & with the entire text matched by the search pattern. This is useful when you want to avoid retyping text:

    :%s/Washington/&, George/

The replacement will say *Washington, George*. The & can also replace a variable pattern (as specified by a regular expression). For example, to surround each line from 1 to 10 with parentheses, type:

    :1,10s/.*/(&)/

The search pattern matches the whole line, and the & "replays" the line, included within your text.

~

The string found is replaced with the replacement text specified in the *last* substitute command. This is useful for repeating an edit. For example, you could say :s/thier/their/ on one line and repeat the change on another line with :s/thier/~/. The search pattern doesn't need to be the same, though. For example, you could say :s/his/their/ on one line and repeat the replacement on another with :s/her/~/.[5]

\u *or* \l

Cause the next character in the replacement string to be changed to uppercase or lowercase, respectively. For example, to change *yes, doctor* into *Yes, Doctor*, you could say:

    :%s/yes, doctor/\uyes, \udoctor/

This is a pointless example, though, since it's easier just to type the replacement string with initial caps in the first place. As with any regular expression, \u and \l are most useful with a variable string. Take, for example, the command we used earlier:

    :%s/\(That\) or \(this\)/\2 or \1/

---

[5] Modern versions of the standard ed editor use % as the sole character in the replacement text to mean "the replacement text of the last substitute command."

The result is *this or That*, but we need to adjust the cases. We'll use `\u` to uppercase the first letter in *this* (currently saved in hold buffer 2), and we'll use `\l` to lowercase the first letter in *That* (currently saved in hold buffer 1):

```
:s/\(That\) or \(this\)/\u\2 or \l\1/
```

The result is *This or that*. Don't confuse the number one with the lowercase `l`; the one comes after.

`\U` *or* `\L` *and* `\e` *or* `\E`

    `\U` and `\L` are similar to `\u` or `\l`, but all following characters are converted to uppercase or lowercase until the end of the replacement string or until `\e` or `\E` is reached. If there is no `\e` or `\E`, all characters of the replacement text are affected by the `\U` or `\L`. For example, to uppercase *Fortran*, you could say:

```
:%s/Fortran/\UFortran/
```

or, using the `&` character to repeat the search string:

```
:%s/Fortran/\U&/
```

All pattern searches are *case sensitive*. That is, a search for *the* does not find *The*. You can get around this by specifying both uppercase and lowercase in the pattern:

```
/[Tt]he
```

You can also instruct the editor to ignore case by typing `:set ic`. See the section for additional details.

More Substitution Tricks

You should know some additional important facts about the substitute command:

- A simple `:s` is the same as `:s//~/`. In other words, repeat the last substitution. This can save enormous amounts of time and typing when you are working your way through a document making the same change repeatedly but you don't want to use a global substitution.

- If you think of the `&` as meaning "the same thing" (as in, what was just matched), this command is relatively mnemonic. You can follow the `&` with a `g` to make the substitution globally on the line, and even use it with a line range:

    `:%&g`   *Repeat the last substitution everywhere*

- The `&` key can be used as a `vi`-mode command to perform the `:&` command, i.e., to repeat the last substitution. This can save even more typing than `:s` ENTER—one keystroke versus three.

- The `:~` command is similar to the `:&` command but with a subtle difference: the search pattern used is the last regular expression used in *any* command, not

necessarily the one used in the last substitute command. For example, in the sequence:

```
:s/red/blue/
:/green
:~
```

the `:~` is equivalent to `:s/green/blue/`.[6]

- Besides the `/` character, you may use any nonalphanumeric, nonspace character as your delimiter, except backslash, double quotes, and the vertical bar (`\`, `"`, and `|`). This is particularly handy when you have to make a change to a pathname. For example:

    ```
    :%s;/user1/tim;/home/tim;g
    ```

- When the `edcompatible` option is enabled, the editor remembers the flags (`g` for global and `c` for confirmation) used on the last substitution and applies them to the next one.

    This is most useful when you are moving through a file and you wish to make global substitutions. You can make the first change:

    ```
    :s/old/new/g
    :set edcompatible
    ```

    and after that, subsequent substitute commands are global.

    Note that, despite the name, no known version of Unix `ed` actually works this way.

Pattern-Matching Examples

Unless you are already familiar with regular expressions, the preceding discussion of special characters probably looks forbiddingly complex. A few more examples should make things clearer. In the examples that follow, a square (□) marks a space; it is not a special character.

Let's work through how you might use some special characters in a replacement. Suppose that you have a long file and that you want to substitute the word *child* with the word *children* throughout that file. You first save the edited buffer with `:w` and then try this global replacement:

```
:%s/child/children/g
```

When you continue editing, you notice occurrences of words such as *childrenish*. You have unintentionally matched the word *childish*. Returning to the last saved buffer with `:e!`, you now try:

---

6 Thanks to Keith Bostic, in the `nvi` documentation, for this example.

```
    :%s/child□/children□/g
```

Note that there is a space after *child*. But this command misses the occurrences *child.*, *child,*, *child:*, and so on. After some thought, you remember that brackets allow you to specify one character from among a list, so you realize a solution:

```
    :%s/child[□,.;:!?]/children[□,.;:!?]/g
```

This searches for *child* followed by either a space (indicated by □) or any one of the punctuation characters `,.;:!?`. You expect to replace this with *children* followed by the corresponding space or punctuation mark, but you've ended up with a bunch of punctuation marks after every occurrence of *children*. You need to save the space and punctuation marks inside a \(and \). Then you can "replay" them with a \1. Here's the next attempt:

```
    :%s/child\([□,.;:!?]\)/children\1/g
```

When the search matches a character inside the \(and \), the \1 on the righthand side restores the same character. The syntax may seem awfully complicated, but this command sequence can save you a lot of work. *Any time you spend learning regular expression syntax will be repaid a thousandfold!*

The command is still not perfect, though. You've noticed that occurrences of *Fairchild* have been changed, so you need a way to match *child* when it isn't part of another word.

As it turns out, `vi` and Vim (but not all other programs that use regular expressions) have a special syntax for saying "only if the pattern is a complete word." The character sequence \< requires the pattern to match at the beginning of a word, whereas \> requires the pattern to match at the end of a word. Using both restricts the match to a whole word. So, in the example task, `\<child\>` finds all instances of the word *child*, whether followed by punctuation or spaces. Here's the substitution command you should use:

```
    :%s/\<child\>/children/g
```

One final possibility is:

```
    :%s/\<child\>/&ren/g
```

Search for General Class of Words

Suppose your subroutine names begin with the prefixes *mgi*, *mgr*, and *mga*:

```
mgibox routine,
mgrbox routine,
mgabox routine,
```

If you want to save the prefixes while changing the name *box* to *square*, either of the following replacement commands does the trick. The first example illustrates how \(and \) can be used to save whatever pattern was actually matched. The second example shows how you can search for one pattern but change another:

```
:g/mg\([ira]\)box/s//mg\1square/g

mgisquare routine,
mgrsquare routine,
mgasquare routine,
```

The global replacement keeps track of whether an *i*, *r*, or *a* is saved. In that way, *box* is changed to *square* only when *box* is part of the routine's name:

```
:g/mg[ira]bo$/s/box/square/g

mgisquare routine,
mgrsquare routine,
mgasquare routine,
```

This has the same effect as the previous command, but it is a little less safe since it could change other instances of *box* on the same line, not just those within the routine names.

Block Move by Patterns

You can also move blocks of text delimited by patterns. For example, assume you have a 150-page reference manual written in a specialized version of XML. Each page is organized into three paragraphs with the same three headings: `<syntax>`, `<description>`, and `<parameters>`. A sample of one reference page follows:

```
<reference>
<description>Get status of named file</description>
<shortname>STAT</shortname>
<syntax>
int stat(const char *filename, struct stat *data);
...
retval = stat(filename, data);</syntax>
<description><para>
Writes the fields of a system data structure into the
structure pointed to by data.
These fields contain (among other
things) information about the file's access
privileges, owner, and time of last modification.
</para></description>
<parameters>
<param><name>filename</name>
<para>A character string variable or constant containing
the Unix pathname for the file whose status you want
to retrieve.
You can give the ...
</para></param></parameters>
</reference>
```

Suppose that you decide to move `<description>` above the `<syntax>` paragraph. With pattern matching, you can move blocks of text on all 150 pages with one command!

```
:g /<syntax>/.,/<description>/-1 move /<parameters>/-1
```

This command works as follows. First, ex finds and marks each line that matches the first pattern (i.e., that contains the word *<syntax>*). Second, for each marked line, it sets `.` (dot, the current line) to that line and executes the command. This command moves the block of lines from the current line (dot) to the line before the one containing the word *<description>* (`/<description>/-1`) to just before the line containing *<parameters>* (`/<parameters>/-1`).[7]

Note that ex can place text only below the line specified.

To tell ex to place text above a line, you first subtract one with `-1`, and then ex places your text below the previous line.

In a case like this, one command saves literally hours of work. This is a real-life example—we once used a pattern match like this to rearrange a reference manual containing hundreds of pages.

Block definition by patterns can be used equally well with other ex commands. For example, if you wanted to delete all `<description>` paragraphs in the reference chapter, you could enter:

```
:g/<description>/,/<parameters>/-1d
```

This very powerful kind of change is implicit in ex's line-addressing syntax, but it is not readily apparent even to experienced users. For this reason, whenever you are faced with a complex, repetitive editing task, take the time to analyze the problem and determine if you can apply pattern-matching tools to get the job done.

More Examples

Since the best way to learn pattern matching is by example, here is a list of pattern-matching examples, with explanations. Study the syntax carefully so that you understand the principles at work. You should then be able to adapt these examples to your own situation.

---

7 We could have moved it to the line after *</description>*, using `move /<\/description>/`. It's not obvious if this is more readable or less readable.

A Word About troff

The standard text formatting tools on Unix are `troff`, for typesetters and laser printers, and its twin brother `nroff`, for terminals and line printers. They accept the same input language.

Input for `troff` consists of text to be formatted intermingled with command lines and escape sequences (such as to italicize or embolden text).

Once upon a time, knowledge of and skill with `troff` and `nroff` were a required part of becoming a "Unix wizard." Over time, their use has fallen off, but they remain necessary for one critical task: authoring manual pages.

So, while we've reduced the number of `troff`-related examples in the book, we haven't removed all of them. We hope the ones that remain will be helpful to you.

1. Put `troff` italicization codes around the word *ENTER*:

       :%s/ENTER/\\fI&\\fP/g

   Notice that two backslashes (\\) are needed in the replacement, because the backslash in the `troff` italicization code will be interpreted as a special character. (\fI alone would be interpreted as *fI*; you must type \\fI to get \fI.)

2. Modify a list of pathnames in a file:

       :%s/\/home\/tim/\/home\/linda/g

   A slash (used as a delimiter in the global replacement sequence) must be escaped with a backslash when it is part of the pattern or replacement; use \/ to get /. An alternate way to achieve this same effect is to use a different character as the pattern delimiter. For example, you could make the previous replacement using colons as delimiters. (The delimiter colons and the `ex` command colon are separate entities.) Thus:

       :%s:/home/tim:/home/linda:g

   This is much more readable.

3. Put HTML italicization codes around the word *ENTER*:

       :%s:ENTER:<I>&</I>:g

   Notice here the use of & to represent the text that was actually matched and, as just described, the use of colons as delimiters instead of slashes.

4. Change all periods to semicolons in lines 1 to 10:

       :1,10s/\./;/g

   A dot has special meaning in regular expression syntax and must be escaped with a backslash (\.).

5. Change all occurrences of the word *help* (or *Help*) to *HELP*:

    `:%s/[Hh]elp/HELP/g`

    or:

    `:%s/[Hh]elp/\U&/g`

    The `\U` changes the pattern that follows to all uppercase. The pattern that follows is the replayed search pattern, which is either *help* or *Help*.

6. Replace *one or more* spaces with a single space:

    `:%s/␣␣*/␣/g`

    Make sure you understand how the asterisk works as a special character. An asterisk following any character (or following any regular expression that matches a single character, such as `.` or `[[:lower:]]`) matches *zero or more* instances of that character. Therefore, you must specify *two* spaces followed by an asterisk to match one or more spaces (one space, plus zero or more spaces).

7. Replace one or more spaces following a colon with exactly two spaces:

    `:%s/:␣␣*/:␣␣/g`

8. Replace one or more spaces following a period *or* a colon with exactly two spaces:

    `:%s/\([:.]\)␣␣*/\1␣␣/g`

    Either of the two characters within brackets can be matched. This character is saved into a hold buffer, using `\(` and `\)`, and restored on the righthand side by the `\1`. Note that within brackets a special character such as a dot does not need to be escaped.

9. Standardize various uses of a word or heading:

    `:%s/^Note[␣:s]*/Notes:␣/g`

    The brackets enclose three characters: a space, a colon, and the letter *s*. Therefore, the pattern `Note[␣s:]` matches *Note␣*, *Notes*, or *Note:*. An asterisk is added to the pattern so that it also matches *Note* (with zero spaces after it) and *Notes:* (the already correct spelling). Without the asterisk, *Note* would be missed entirely and *Notes:* would be incorrectly changed to *Notes:␣:*. Simultaneously, this truncates multiple spaces down to one, so that *Note:␣␣* becomes *Notes:␣*.

10. Delete all empty lines:

    `:g/^$/d`

    What you are actually matching here is the beginning of the line (`^`) followed by the end of the line (`$`), with nothing in between.

11. Delete all empty lines, plus any lines that contain only whitespace:

    `:g/^[␣tab]*$/d`

In the example, a tab is shown as *tab*. A line may appear to be empty but may in fact contain spaces or tabs. The previous example does not delete such a line. This example, like the previous one, searches for the beginning and end of the line. But instead of having nothing in between, the pattern tries to find any number of spaces or tabs. If no spaces or tabs are matched, the line is empty. To delete lines that contain whitespace but that *aren't* empty, you would have to match lines with *at least* one space or tab:

`:g/^[□tab][□tab]*$/d`

12. Delete all leading spaces on every line:

    `:%s/^□□*\(.*\)/\1/`

    Use `^□□*` to search for one or more spaces at the beginning of each line; then use `\(.*\)` to save the rest of the line into the first hold buffer. Restore the line without leading spaces, using `\1`. This can be done more simply with `s/^□□*//`.

13. Delete all spaces at the end of every line:

    `:%s/□□*$//`

    For each line, remove one or more spaces at the end of the line.

    Because of the `^` and `$` anchors, the substitutions in this example and the previous one happen only once on any given line, so the `g` option doesn't need to follow the replacement string.

14. Insert a `//□` at the start of each line from the current line to the next line that starts with `}`:

    `:.,/^}/s;^;//□;`

    What we're really doing here is "replacing" the start of the line with `//□`. Of course, the start of the line (being a logical construct, not an actual character) isn't really replaced!

    What this does is comment-out all lines from the current line (dot) to the next line that starts with a right (closing) brace, using C++ `//` comments. Typically, you'd use this by placing the cursor at the first line of a function definition to comment-out an entire function.

    Note the use of a semicolon as the delimiter for the substitute command when the substitution text contains one or more slashes.

15. Add a period to the end of the next six lines:

    `:.,+5s/$/./`

    The line address indicates the current line plus five lines. The `$` indicates the end of line. As in the previous example, the `$` is a logical construct. You aren't really replacing the end of the line.

16. Reverse the order of all hyphen-separated items in a list:

```
:%s/\(.*\)□-□\(.*\)/\2□-□\1/
```

Use `\(.*\)` to save text on the line into the first hold buffer, but only until you find □-□. Then use `\(.*\)` to save the rest of the line into the second hold buffer. Restore the saved portions of the line, reversing the order of the two hold buffers. The effect of this command on several items is shown here:

```
more - display files
```

becomes:

```
display files - more
```

and:

```
lp - print files
```

becomes:

```
print files - lp
```

This can be done even more succinctly:

```
:%s/\(.*\)\(□-□\)\(.*\)/\3\2\1/
```

17. Change every letter in a file to uppercase:

```
:%s/.*/\U&/
```

or:

```
:%s/./\U&/g
```

The `\U` flag at the start of the replacement string tells the editor to change the replacement to uppercase. The `&` character replays the text matched by the search pattern as the replacement.

These two commands are equivalent; however, the first form is considerably faster, since it results in only one substitution per line (`.*` matches the entire line, once per line), whereas the second form results in repeated substitutions on each line (`.` matches only a single character, with the replacement repeated on account of the trailing `g`).

18. Reverse the order of lines in a file:[8]

```
:g/.*/mo0
```

The search pattern matches all lines (a line contains zero or more characters). Each line is moved, one by one, to the top of the file (that is, moved after imaginary line zero). As each matched line is placed at the top, it pushes the previously moved lines down, one by one, until the last line is on top. Since all lines have a beginning, the same result can be achieved more succinctly:

```
:g/^/mo0
```

---

8 From an article by Walter Zintz in *UnixWorld*, May 1990.

19. In a text-file database, on all lines not marked *Paid in full*, append the phrase *Overdue*:

    `:g!/Paid in full/s/$/ Overdue/`

    or the equivalent:

    `:v/Paid in full/s/$/ Overdue/`

    To affect all lines *except* those matching your pattern, you add a ! to the `:g` command, or you can just use the v command.

20. For any line that doesn't begin with a number, move the line to the end of the file:

    `:g!/^[[:digit:]]/m$`

    or:

    `:g/^[^[:digit:]]/m$`

    As the first character within brackets, a caret negates the sense, so the two commands have the same effect. The first one says, "Don't match lines that begin with a number," and the second one says, "Match lines that don't begin with a number."

    Note that there *is* a rather subtle difference between the commands. The first one affects empty lines; the second one does not. How so? `/^[[:digit:]]/` matches lines that start with a digit. The ! after the `:g` negates that, matching lines that don't start with a digit. This includes empty lines. However, `/^[^[:digit:]]/` matches lines that start with a nondigit character; in order to match, there has to be a character present on the line.

21. Change manually numbered section heads (e.g., 1.1, 1.2, etc.) to HTML `<h1>` heading markers:

    `:%s;^[1-9]\.[1-9] \(.*\);<h1>\1</h1>;`

    The search string matches a digit other than zero, followed by a period, followed by another nonzero digit, followed by a space, followed by anything. The command just shown won't find chapter numbers containing two or more digits. To do so, modify the command like this:

    `:%s;^[1-9][0-9]*\.[1-9] \(.*\);<h1>\1</h1>;`

    Now it matches chapters 10 to 99 (digits 1 to 9, followed by a digit), 100 to 999 (digits 1 to 9, followed by two digits), and so on. The command still finds chapters 1 to 9 (digits 1 to 9, followed by no digit) as well.

22. Remove numbering from section headings in a document. You want to change the sample lines:

    ```
    2.1 Introduction
    10.3.8 New Functions
    ```

    into the lines:

```
Introduction
New Functions
```

Here's the command to do this:

```
:%s/^[1-9][0-9]*\.[1-9][0-9.]*□//
```

The search pattern resembles the one in the previous example, but now the numbers vary in length. At a minimum, the headings contain *number*, *period*, *number*, so you start with the search pattern from the previous example:

```
[1-9][0-9]*\.[1-9]
```

But in this example, the heading may continue with any number of digits or periods:

```
[0-9.]*
```

23. Change the word *Fortran* to the phrase *FORTRAN (acronym of FORmula TRANslation)*:

```
:%s/\(For\)\(tran\)/\U\1\2\E□(acronym□of□\U\1\Emula□\U\2\Eslation)/g
```

First, since we notice that the words *FORmula* and *TRANslation* use portions of the original words, we decide to save the search pattern in two pieces: `\(For\)` and `\(tran\)`. The first time we restore it, we use both pieces together, converting all characters to uppercase: `\U\1\2`. Next, we undo the uppercase with `\E`; otherwise, the remaining replacement text would all be uppercase. The replacement continues with actual typed words, and then we restore the first hold buffer. This buffer still contains *For*, so again we convert to uppercase first: `\U\1`. Immediately after, we lowercase the rest of the word: `\Emula`. Finally, we restore the second hold buffer. This contains *tran*, so we precede the "replay" with uppercase, follow it with lowercase, and type out the rest of the word: `\U\2\Eslation)`.

A Final Look at Pattern Matching

We conclude this chapter by presenting sample tasks that involve complex pattern-matching concepts. Rather than solve the problems right away, we'll work toward the solutions step by step.

Deleting an Unknown Block of Text

Suppose you have a few lines with this general form:

```
the best of times; the worst of times:  moving
The coolest of times; the worst of times:  moving
```

The lines that you're concerned with always end with *moving*, but you never know what the first two words might be. You want to change any line that ends with *moving* to read:

```
The greatest of times; the worst of times:  moving
```

Since the changes must occur on certain lines, you need to specify a context-sensitive global replacement. Using `:g/moving$/` matches lines that end with *moving*. Next, you realize that your search pattern could be any number of any character, so the metacharacters `.*` come to mind. But these match the whole line unless you somehow restrict the match. Here's your first attempt:

```
:g/moving$/s/.*of/The␣greatest␣of/
```

This search string, you decide, will match from the beginning of the line to the first *of*. Since you needed to specify the word *of* to restrict the search, you simply repeat it in the replacement. Here's the resulting line:

```
The greatest of times:  moving
```

Something went wrong! The replacement gobbled the line up to the second *of* instead of the first. Here's why: when given a choice, the action of "match any number of any character" matches *as much text as possible*.[9] In this case, since the word *of* appears twice, your search string finds:

```
the best of times; the worst of
```

rather than:

```
the best of
```

Your search pattern needs to be more restrictive:

```
:g/moving$/s/.*of␣times;/The␣greatest␣of␣times;/
```

Now the `.*` matches all characters up to the instance of the phrase *of times;*. Since there's only one instance, it has to be the first.

There are cases, though, when it is inconvenient, or even incorrect, to use the `.*` metacharacters. For example, you might find yourself typing many words to restrict your search pattern, or you might be unable to restrict the pattern by specific words (if the text in the lines varies widely). The next section presents such a case.

Switching Items in a Textual Database

Suppose you want to switch the order of all last names and first names in a (text) database. The lines look like this:

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 765-4321
Name: Joy, Susan S.; Areas: Graphics; Phone: 999-3333
```

Each field name ends with a colon, and each field is separated by a semicolon. Using the top line as an example, you want to change *Feld, Ray* to *Ray Feld*. We'll present

---

9 More formally, the *longest, leftmost* text is what matches.

some commands that look promising but don't work. After each command, we show you the line the way it looked before the change and after the change:

```
:%s/: \(.*\), \(.*\);/: \2 \1;/
```

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 765-4321        Before
Name: Unix Feld, Ray; Areas: PC; Phone: 765-4321         After
```

We've highlighted the contents of the first hold buffer in **bold** and the contents of the second hold buffer in *italic*. Note that the first hold buffer contains more than you want. Since it was not sufficiently restricted by the pattern that follows it, the hold buffer was able to save up to the second comma. Now you try to restrict the contents of the first hold buffer:

```
:%s/: \(....\), \(.*\);/: \2 \1;/
```

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 765-4321        Before
Name: Ray; Areas: PC, Unix Feld; Phone: 765-4321         After
```

Here you've managed to save the last name in the first hold buffer, but now the second hold buffer saves anything up to the last semicolon on the line. Now you restrict the second hold buffer, too:

```
:%s/: \(....\), \(...\);/: \2 \1;/
```

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 765-4321        Before
Name: Ray Feld; Areas: PC, Unix; Phone: 765-4321         After
```

This gives you what you want, but only in the specific case of a four-letter last name and a three-letter first name. (The previous attempt included the same mistake.) Why not just return to the first attempt, but this time be more selective about the end of the search pattern?

```
:%s/: \(.*\), \(.*\); Area/: \2 \1; Area/
```

```
Name: Feld, Ray; Areas: PC, Unix; Phone: 765-4321        Before
Name: Ray Feld; Areas: PC, Unix; Phone: 765-4321         After
```

This works, but we'll continue the discussion by introducing an additional concern. Suppose that the *Area* field isn't always present or isn't always the second field. The command just shown won't work on such lines.

We introduce this problem to make a point. Whenever you rethink a pattern match, it's usually better to work toward refining the variables (the metacharacters), rather than using specific text to restrict patterns. The more variables you use in your patterns, the more powerful your commands will be.

In the current example, think again about the patterns you want to switch. Each word starts with an uppercase letter and is followed by any number of lowercase letters, so you can match the names like this:

```
[[:upper:]][[:lower:]]*
```

A last name might also have more than one uppercase letter (*McFly*, for example), so you'd want to search for this possibility in the second and succeeding letters:

```
[[:upper:]][[:alpha:]]*
```

It doesn't hurt to use this for the first name, too (you never know when *McGeorge Bundy* will turn up). Your command now becomes:

```
:%s/: \([[:upper:]][[:alpha:]]*\), \([[:upper:]][[:alpha:]]*\);/: \2 \1;/
```

Quite forbidding, isn't it? It still doesn't cover the case of a name like *Joy, Susan S.* Since the first-name field might include a middle initial, you need to add a space and a period within the second pair of brackets. But enough is enough. Sometimes, specifying exactly what you want is more difficult than specifying what you *don't* want. In your sample database, the last names end with a comma, so a last-name field can be thought of as a string of characters that are *not* commas:

```
[^,]*
```

This pattern matches characters up until the first comma. Similarly, the first-name field is a string of characters that are *not* semicolons:

```
[^;]*
```

Putting these more efficient patterns back into your previous command, you get:

```
:%s/: \([^,]*\), \([^;]*\);/: \2 \1;/
```

The same command could also be entered as a context-sensitive replacement. If all lines begin with *Name*, you can say:

```
:g/^Name/s/: \([^,]*\), \([^;]*\);/: \2 \1;/
```

You can also add an asterisk after the first space, in order to match a colon that has extra spaces (or no spaces) after it:

```
:g/^Name/s/: *\([^,]*\), \([^;]*\);/: \2 \1;/
```

Using :g to Repeat a Command

In the usual way we've seen the `:g` command used, it selects lines that are typically then edited by subsequent commands on the same **ex** command line—for example, we select lines with `:g` and then make substitutions on them, or we select them and delete them:

```
:g/mg[ira]box/s/box/square/g
:g/^$/d
```

However, in his two-part tutorial in *UnixWorld*,[10] Walter Zintz makes an interesting point about the `:g` command. This command selects lines, but the associated editing commands need not actually affect the lines that are selected.

Instead, he demonstrates a technique by which you can repeat `ex` commands some arbitrary number of times. For example, suppose you want to place 10 copies of lines 12 through 17 of your file at the end of your current file. You could type:

```
:1,10g/^/ 12,17t$
```

This is a very unexpected use of `:g`, but it works! The `:g` command selects line 1, executes the specified `t` command, and then goes on to line 2 to execute the next copy command. When line 10 is reached, `ex` will have made 10 copies.

Collecting Lines

Here's another advanced `:g` example, again building on suggestions provided in Zintz's tutorial. Suppose you're editing a document that consists of several parts. Part 2 of this file is shown here, using ellipses to show omitted text and displaying line numbers for reference:

```
301  Part 2
302  Capability Reference
303  .LP
304  Chapter 7
305  Introduction to the Capabilities
306  This and the next three chapters ...

400  ... and a complete index at the end.
401  .LP
402  Chapter 8
403  Screen Dimensions
404  Before you can do anything useful
405  on the screen, you need to know ...

555  .LP
556  Chapter 9
557  Editing the Screen
558  This chapter discusses ...

821  .LP
822  Part 3
823  Advanced Features
824  .LP
825  Chapter 10
826  ....
```

---

10 Part 1, "`vi` Tips for Power Users," appears in the April 1990 issue of *UnixWorld*; Part 2, "Using `vi` to Automate Complex Edits," appears in the May 1990 issue. The examples presented are from Part 2. The tutorial is available in this book's GitHub repository.

Each chapter number appears on one line, the chapter title appears on the line below, and the chapter text (marked in **bold** for emphasis) begins on the line below that. The first thing you'd like to do is copy the beginning line of each chapter, sending it to an already existing file called *begin*.

Here's the command that does this:

```
:g /^Chapter/ .+2w >> begin
```

You must be at the top of your file before issuing this command. First, you search for *Chapter* at the start of a line, but then you want to run the command on the beginning line of each chapter—the second line below *Chapter*. Because a line beginning with *Chapter* is now selected as the current line, the line address `.+2` indicates the second line below it. The equivalent line addresses `+2` or `++` work as well. You want to write these lines to an existing file named *begin*, so you issue the `w` command with the append operator `>>`.

Suppose you want to send the beginnings of chapters that are only within Part 2. You need to restrict the lines selected by `:g`, so you change your command to this:

```
:/^Part 2/,/^Part 3/g /^Chapter/ .+2w >> begin
```

Here, the `:g` command selects the lines that begin with *Chapter*, but it searches only that portion of the file from a line starting with *Part 2* through a line starting with *Part 3*. If you issue the command just shown, the last lines of the file *begin* will read as follows:

```
This and the next three chapters ...
Before you can do anything useful
This chapter discusses ...
```

These are the lines that begin Chapters 7, 8, and 9.

In addition to the lines you've just sent, you'd like to copy chapter titles to the end of the document, in preparation for making a table of contents. You can use the vertical bar to tack on a second command after your first command, like so:

```
:/^Part 2/,/^Part 3/g /^Chapter/ .+2w >> begin | +t$
```

Remember that with any subsequent command, line addresses are relative to the previous command. The first command has marked lines (within Part 2) that start with *Chapter*, and the chapter titles appear on a line below such lines. Therefore, to access chapter titles in the second command, the line address is `+` (or the equivalents `+1` or `.+1`). Then use `t$` to copy the chapter titles to the end of the file.

As these examples illustrate, thought and experimentation may lead you to some unusual editing solutions. Don't be afraid to try things. Just be sure to back up your file first! Of course, with the infinite "undo" facility in Vim, you may not even need to save a backup copy; see the section "Extended Undo" on page 180 for more information.

Advanced Editing

This chapter introduces you to some of the more advanced capabilities of `vi` and Vim and the `ex` editor underlying them. You should be reasonably familiar with the material presented in the earlier chapters of this book before you start working with the concepts presented here.

As we did in earlier chapters, this chapter presents facilities common to all versions of `vi`, but in the context of Vim. When you see "`vi`" by itself here, you can generally read it as "`vi` and Vim."

We have divided this chapter into five parts. The first part discusses a number of ways to set options that allow you to customize your editing environment. You'll learn how to use the `set` command and how to create a number of different editing environments using *.exrc* files.

The second part discusses how you can execute Unix commands from within the editor, and how you can use the editor to filter text through Unix commands.

The third part discusses various ways to save long sequences of commands by reducing them to abbreviations, or even to commands that use only one keystroke (this is called *mapping* keys). It also includes a section on @-functions, which allow you to store command sequences in a register.

The fourth part discusses the use of `ex` scripts from the Unix command line or from within shell scripts. Scripting provides a powerful way to make repetitive edits.

The fifth part discusses some features that are especially useful to programmers. There are options that control line indentation and an option to display invisible characters (specifically tabs and newlines). There are search commands that are useful with program code blocks or with C and C++ functions.

Customizing vi and Vim

`vi` and Vim operate differently on various terminals.

On modern Unix systems, the editor gets operating instructions about your terminal type from the `terminfo` terminal database.[1]

There are also a number of options that you can set from within the editor that affect how it operates. For example, you can set a right margin that causes `vi` to wrap lines automatically so you don't need to hit ENTER.

You change options from within the editor by using the `ex` command `:set`. In addition, whenever `vi` and Vim start up, they read a file in your home directory called *.exrc* for further operating instructions. By placing `:set` commands in this file, you can modify the way the editor acts whenever you use it.

You can also set up *.exrc* files in local directories to initialize various options that you want to use in different environments. For example, you might define one set of options for editing English text, and another set for editing source programs. The *.exrc* file in your home directory is executed first, and then the one in your current directory.

Finally, any commands stored in the environment variable `EXINIT` are executed on startup. The settings in `EXINIT` take precedence over those in the home directory *.exrc* file.

The :set Command

There are two types of options that can be changed with the `:set` command: toggle options, which are either on or off, and options that take a numeric or string value (such as the location of a margin or the name of a file).

Toggle options may be on or off by default. To turn a toggle option on, the command is:

```
:set option
```

To turn a toggle option off, the command is:

```
:set nooption
```

For example, to specify that pattern searches should ignore case, type:

```
:set ic
```

---

1 The location of this database varies from vendor to vendor. Try the command `man terminfo` to get more information about your specific system.

If you want `vi` to return to being case sensitive in searches, give the command:

    :**set** noic

Many options have both complete names and abbreviations. In the previous example, `ic` is short for `ignorecase`; you could also have entered `set ignorecase` to ignore case, and `set noignorecase` to restore the default behavior.

Vim lets you toggle the value of an option with:

    :set *option*!

Some options have a value assigned to them. For example, the `window` option sets the number of lines shown in the screen's "window." You set values for these options with an equals sign (=):

    :**set** window=20

During an editing session, you can check which options are in use. The command:

    :**set all**

displays the complete list of options, including options that you have set and defaults that the editor has "chosen."

The display should look something like this:[2]

autoindent	nomodelines	noshowmode
autoprint	nonumber	noslowopen
noautowrite	open	nosourceany
nobeautify	nooptimize	tabstop=8
directory=/var/tmp	paragraphs=IPLPPPQPP LIpplpipbp	taglength=0
noedcompatible	prompt	tags=tags /usr/lib/tags
noerrorbells	noreadonly	term=xterm
noexrc	redraw	noterse
flash	remap	timeout
hardtabs=8	report=5	ttytype=xterm
noignorecase	scroll=11	warn
nolisp	sections=NHSHH HUnhsh	window=23
nolist	shell=/bin/bash	wrapscan
magic	shiftwidth=8	wrapmargin=0
mesg	showmatch	nowriteany

You can find out the current value of any individual option by name, using the command:

    :set *option*?

The command:

    :**set**

_____

2 The result of `:set all` depends very much on the version of `vi` you have. This particular display is typical of Unix `vi`. The order is alphabetical going down the columns, ignoring any leading `no`. Vim has *many* more options than what is shown here.

shows options that you have specifically changed, or set, either in your *.exrc* file or during the current session. For example, the display might look like this:

```
number sect=AhBhChDh window=20 wrapmargin=10
```

The .exrc File

The *.exrc* file that controls your own editing environment is in your home directory. You can modify the *.exrc* file with Vim, just as you can any other text file. (Of course, any new settings don't take effect until you restart Vim or explicitly reread the file with the `:source` command.)

If you don't yet have an *.exrc* file, simply create one as you would any other file. Enter into this file the `set`, `ab`, and `map` commands that you want to have in effect whenever you edit. (`ab` and `map` are discussed later in this chapter.) A sample *.exrc* file might look like this:

```
set nowrapscan wrapmargin=7
set sections=SeAhBhChDh nomesg
map q :w^M:n^M
ab ORA O'Reilly Media, Inc.
```

Since the file is actually read by `ex` before it enters visual mode (`vi`), commands in *.exrc* need not have a preceding colon.

Alternate Environments

In addition to reading the *.exrc* file in your home directory, the editor will read a file called *.exrc* in the current directory. This lets you set options that are appropriate to a particular project.

In all modern versions of `vi`, including Vim, you have to first set the `exrc` option in your home directory's *.exrc* file before the editor will read the *.exrc* file in the current directory:

```
set exrc
```

This mechanism prevents other people from placing an *.exrc* file into your working directory whose commands might jeopardize the security of your system.[3]

For example, you might want to have one set of options in a directory mainly used for programming:

```
set number autoindent sw=4 terse
set tags=/usr/lib/tags
```

and another set of options in a directory used for text editing:

---

3 The original versions of `vi` automatically read both files, if they existed. The `exrc` option closes a potential security hole.

```
set wrapmargin=15 ignorecase
```

Note that you can set certain options in the *.exrc* file in your home directory and unset them in a local directory.

You can also define alternate editing environments by saving option settings in a file other than *.exrc* and reading in that file with the `:so` command. (`so` is short for `source`.)

For example:

```
:so .progoptions
```

The editor does not use a search path to find files for `:so`. Thus, filenames that do not start with a `/` are considered to be relative to the current directory.

Local *.exrc* files are also useful for defining abbreviations and key mappings (described later in this chapter). Authors using a markup language to write a book or other document can easily save all the abbreviations to be used in that book in an *.exrc* file in the directory in which the book is being created.

Note that this assumes all the book's files are in the same directory. If they are divided among subdirectories, you'd have to copy the *.exrc* file to each subdirectory, or do something different, like using Vim's `autocmd` feature, which allows you to set options or perform actions based on the filename extension of the file you're editing. This makes it easy to customize your editing one way for, say, DocBook XML, and another way for, say, AsciiDoc or LaTeX. See the section "Autocommands" on page 300.

Some Useful Options

As you can see when you type `:set all`, there are an awful lot of options that can be set. Many of them are used internally by the editor and aren't usually changed. Others are important in certain cases but not in others (for example, `noredraw` and `window` can be useful over a cross-continental `ssh` session). Table B-1 in the section "Heirloom and Solaris vi Options" on page 461 contains a brief description of each option. We recommend that you take some time to play with setting options. If an option looks interesting, try setting it (or unsetting it) and watch what happens while you edit. You may find some surprisingly useful tools.

As discussed earlier in the section "Movement Within a Line" on page 20, one option, `wrapmargin`, is essential for editing nonprogram text. `wrapmargin` specifies the size of the right margin that is used to autowrap text as you type. (This saves manually typing carriage returns.)[4] A typical value is 7 to 15:

---

4 On a computer, entering a *carriage return* means pressing the ENTER key. The term comes from typewriters, where, upon finishing a line, you used a lever to shift the paper up one line and *return* the carriage (the part

```
    :set wrapmargin=10
```

Three other options control how the editor acts when conducting a search. Normally, a search differentiates between uppercase and lowercase (*foo* does not match *Foo*), wraps around to the beginning of the file (meaning that you can begin your search anywhere in the file and still find all occurrences), and recognizes wildcard characters when pattern matching. The default settings that control these options are `ignorecase`, `wrapscan`, and `magic`, respectively. To change any of these defaults, you would set the opposite toggle options: `noignorecase`, `nowrapscan`, and `nomagic`.

Options that may be of particular interest to programmers include `autoindent`, `expandtab`, `list`, `number`, `shiftwidth`, `showmatch`, and `tabstop`, as well as their opposite toggle options.

Finally, consider using the `autowrite` option. When set, the editor automatically writes out the contents of a changed buffer when you issue the `:n` (next) command to move to the next file to be edited, and before running a shell command with `:!`.

Executing Unix Commands

You can display or read in the results of any Unix command while you are editing. An exclamation mark (`!`) tells `ex` to create a shell and to regard what follows as a Unix command:

```
    :!command
```

So if you are editing and you want to double-check the current directory without exiting `vi`, you can enter:

```
    :!pwd
```

The full path of the current directory appears on your screen; press ENTER to continue editing at the same place in your file.

If you want to give several Unix commands in a row without returning to the editing session in between, you can create a shell with the following `ex` command:

```
    :sh
```

When you want to exit the shell and return to `vi`, press CTRL-D. (This works even from within `gvim`, the GUI version of Vim.)

---

holding the paper) back to the beginning of the line. This is the genesis of the ASCII characters LF and CR (linefeed and carriage return).

You can combine `:read` with a call to the shell to read the results of a Unix command into your file. As a very simple example:

```
:read !date
```

or more simply:

```
:r !date
```

reads the system's date information into the text of your file. By preceding the `:r` command with a line address, you can read the result of the command in at any desired point in your file. By default, it is brought in after the current line.

Suppose you are editing a file and want to read in four phone numbers from a file called *phone*, but in alphabetical order. *phone* reads:

```
Willing, Sue   333-4444
Walsh, Linda   555-6666
Quercia, Valerie   777-8888
Dougherty, Nancy   999-0000
```

The command:

```
:r !sort phone
```

reads in the contents of *phone* after they have been passed through the `sort` filter:

```
Dougherty, Nancy   999-0000
Quercia, Valerie   777-8888
Walsh, Linda   555-6666
Willing, Sue   333-4444
```

Suppose you are editing a file and want to insert text from another file in the directory, but you can't remember the new file's name. You *could* perform this task the long way: exit your file, give the `ls` command, note the correct filename, reenter your file, and search for your place.

Or you could do the task in fewer steps:

Keystrokes	Results
`:!ls`	file1 file2 letter newfile practice
	Display a list of files in the current directory. Note the correct filename. Press ENTER to continue editing.
`:r newfile`	"newfile" 35L, 1569C 2,1 Top
	Read in the new file.

One of the authors very often combines the `r` command with `%` as the filename to make it easy to correct spelling errors in a document:

```
:w
:$r !spell %
```

This saves the file and then reads the output of the `spell` command on the file into the buffer at the end. (On some systems you might want to use `:r !spell % | sort -u` to get a sorted list of misspelled words.)

With the list of misspelled words in the buffer, our author then goes through them one by one, searching through the file and correcting errors, and deleting each word from the list when done.

Filtering Text Through a Command

You can also send a block of text as standard input to a Unix command. The output from this command replaces the block of text in the buffer. You can filter text through a command from either `ex` or `vi`. The main difference between the two methods is that you indicate the block of text with line addresses in `ex` and with text objects (movement commands) in `vi`.

Filtering text with ex

The first example demonstrates how to filter text with `ex`. Assume that the list of names in the preceding example, instead of being contained in a separate file called *phone*, is already contained in the current file on lines 96 through 99. You simply type the addresses of the lines you want to filter, followed by an exclamation mark and the shell command to be executed. For example, the command:

```
:96,99!sort
```

passes lines 96 through 99 through the `sort` filter and replaces those lines with the output of `sort`.

Filtering text with vi motion commands

In `vi` mode, text is filtered through a Unix command by typing an exclamation mark followed by any of the `vi` movement keystrokes that indicate a block of text, and then by the shell command line to be executed. For example:

```
!)command
```

passes the next sentence through *command*.

There are a few unusual aspects of the way `vi` acts when you use this feature:

- The exclamation mark doesn't appear on your screen right away. When you type the keystroke(s) for the text object you want to filter, the exclamation mark appears at the bottom of the screen, *but the character you type to reference the object does not.*

- Text blocks must be more than one line, so you can use only the keystrokes that move more than one line (`G`, `{ }`, `( )`, `[[ ]]`, `+`, `-`). To repeat the effect, a number may precede either the exclamation mark or the text object. (For example, both `!10+` and `10!+` indicate the next 10 lines.) Objects such as `w` do not work unless enough of them are specified so as to exceed a single line. You can also use a slash (`/`) followed by a pattern and a carriage return to specify the object. This takes the text up to the pattern as input to the command.

- Entire lines are affected. For example, if your cursor is in the middle of a line and you issue a command to go to the end of the next sentence, the entire lines containing the beginning and end of the sentence are changed, not just the sentence itself.[5]

- There is a special text object that can be used only with this command syntax; you can specify the current line by entering a second exclamation mark:

      !!*command*

  Remember that either the entire sequence or the text object can be preceded by a number to repeat the effect. For instance, to change lines 96 through 99 as in the previous example, you could position the cursor on line 96 and enter either:

      4!!**sort**

  or:

      !4!**sort**

  As another example, assume you have a portion of text in a file that you want to change from lowercase to uppercase letters. You could process that portion with the `tr` command to change the case. In this example, the second sentence is the block of text to be filtered through the command:

      One sentence before.
      With a screen editor you can scroll the page
      move the cursor, delete lines, insert characters,
      and more, while seeing the results of your edits
      as you make them.
      One sentence after.

---

5 Of course, there's always an exception. In this example, Vim changes only the current line.

Keystrokes	Results
| `!)` | ```
One sentence after.
~
~
~
.,.+4!▮
``` |
| | Line numbers and an exclamation mark appear on the last line to prompt you for the shell command. The `)` indicates that a sentence is the unit of text to be filtered. |
| `tr '[:lower:]' '[:upper:]'` | ```
One sentence before.
▮ITH A SCREEN EDITOR YOU CAN SCROLL THE PAGE
MOVE THE CURSOR, DELETE LINES, INSERT CHARACTERS,
AND MORE, WHILE SEEING THE RESULTS OF YOUR EDITS
AS YOU MAKE THEM.
One sentence after.
``` |
| | Enter the shell command and press ENTER. The input is replaced by the output. |

To repeat the previous command, the syntax is:

> `! object !`

It is sometimes useful when editing electronic mail to filter your text through the `fmt` program to "beautify" it before sending the message. Remember that the "original" input is replaced by the output. Fortunately, if there is a mistake—such as an error message being sent instead of the expected output—you can undo the command and restore the lines.

Saving Commands

Often you type the same long phrases over and over in a file. There are a number of different ways of saving long sequences of commands, both in command mode and in insert mode. When you call up one of these saved sequences to execute it, all you do is type a few characters (or even only one), and the entire sequence is executed as if you had entered the whole sequence of commands one by one.

Word Abbreviation

You can define abbreviations that the editor will automatically expand into the full text whenever you type the abbreviation in insert mode. To define an abbreviation, use this `ex` command:

> `:ab abbr phrase`

abbr is an abbreviation for the specified *phrase*. The sequence of characters that make up the abbreviation is expanded in insert mode only if you type it as a full word; *abbr* is not expanded within a word.

Suppose in the file *practice* you want to enter text that contains a frequently recurring phrase, such as a difficult product or company name. The command:

    :**ab** imrc International Materials Research Center

abbreviates *International Materials Research Center* to the initials *imrc*. Now whenever you type *imrc* in insert mode, *imrc* expands to the full text:

Keystrokes	Results
ithe imrc	the International Materials Research Center

Abbreviations expand as soon as you press a nonalphanumeric character (e.g., punctuation), a space, a carriage return, or ESC (returning to command mode). When you are choosing abbreviations, choose combinations of characters that don't ordinarily occur while you are typing text. If you create an abbreviation that ends up expanding in places where you don't want it to, you can disable the abbreviation by typing:

    :unab *abbr*

(The abbreviation is expanded when you hit ENTER for the :unab command, but don't worry—it is still disabled correctly.) To list your currently defined abbreviations, type:

    :**ab**

The characters that compose your abbreviation cannot also appear at the end of your phrase. For example, if you issue the command:

    :**ab** PG This movie **is** rated PG

vi gives you the message "No tail recursion," and the abbreviation won't be set. The message means that you have tried to define something that will expand itself repeatedly, creating an infinite loop. If you issue the command:

    :**ab** PG the PG rating system

you won't get a warning message.

When tested, we obtained the following results on these vi versions:

Solaris /usr/xpg7/bin/vi *and "Heirloom"* vi
 The tail recursive version is not allowed, while the version with the name in the middle of the expansion expands only once.

Vim
 Both forms are detected and expand only once.

If you are using Unix vi, we recommend that you test your version before repeating your abbreviation as part of the defined phrase.

Using the map Command

While you're editing, you may find that you are using a command sequence frequently, or that you occasionally use a very complex command sequence. To save yourself keystrokes, or the time that it takes to remember the sequence, you can assign the sequence to an unused key by using the `map` command.

The `map` command acts a lot like `ab` except that you define a macro for command mode instead of for insert mode:

`:map x sequence`
> Define character *x* as a *sequence* of editing commands.

`:unmap x`
> Disable the *sequence* defined for *x*.

`:map`
> List the characters that are currently mapped.

Before you can start creating your own maps, you need to know the keys that `vi` doesn't use in command mode that are available for user-defined commands:

Letters
> g, K, q, V, and v

Control keys
> ^A, ^K, ^O, ^W, and ^X

Symbols
> _, *, \, and =

Vim does use all of these characters, except for ^K, ^_, and \.

> The = is used by `vi` if Lisp mode is set, and to do text formatting by Vim. In many modern versions of `vi`, the _ is equivalent to the ^ command, and Vim has a "visual mode" that uses the v, V, and ^V keys. (See the section "Visual Mode Motion" on page 175.) The moral is to test your version carefully.

Depending on your terminal, you may also be able to associate map sequences with special function keys. You can also map keys that the command mode already uses, but in that case you lose access to the key's default function; we show an example later, in the section "More Examples of Mapping Keys" on page 128. The section "Several Convenience Maps" on page 339 provides some additional, more heavy-duty mapping examples.

With maps, you can create simple or complex command sequences. As a simple example, you could define a command to reverse the order of words. In `vi`, with the cursor as shown:

```
you can █he scroll page
```

the sequence to put *the* after *scroll* would be `dwelp`: delete word, `dw`; move to the end of next word, `e`; move one space to the right, `l`; put the deleted word there, `p`. Saving this sequence:

```
:map v dwelp
```

enables you to reverse the order of two words at any time in the editing session with the single keystroke v.

Mapping with a Leader

Vim uses almost every key for something. This can make it tricky or confusing to decide what key(s) to map. Therefore, Vim provides an alternative way to map by letting you define a map with the special variable `mapleader`. By default `mapleader` has the value \ (a backslash).

Now you can define a key mapping without picking obscure, unused Vim keys or sacrificing an existing Vim key/action. When defined with the mapleader, you simply type the leader character and then the key defined in the map.

For example, suppose you want to create a mnemonic to *quit* Vim and choose `q` as the easy-to-remember key, but you don't want to give up using `q` as the starting character for certain multicharacter Vim commands (e.g., `qq` to start recording a macro). To do so, map `q` with the leader. Here's how to do it:

```
:map <leader>q :q<cr>
```

You can now execute the `ex` command `:quit` with the keystrokes `\q`.

Set `mapleader` to your favorite choice if you prefer something other than `\`. Since `mapleader` is a Vim variable, the syntax to do so is:

```
:let mapleader="X"
```

where *X* is your leader character of choice.

Protecting Keys from Interpretation by ex

Note that when defining a map, you cannot simply type certain keys, such as ENTER, ESC, BACKSPACE, and DELETE, as part of the command to be mapped, because these keys already have meaning within `ex`. If you want to include one of these keys as part of the command sequence, you must escape the key's normal meaning by preceding it with CTRL-V. The keystroke `^V` appears in the map as the `^`

character. Characters following the `^V` also do not appear as you expect. For example, a carriage return appears as `^M`, escape as `^[`, backspace as `^H`, and so on.

On the other hand, if you want to use a control character as the character to be mapped, in most cases all you have to do is hold down the CTRL key and press the letter key at the same time. So, for example, all you need to do to map `^A` is to type:

    :map CTRL-A *sequence*

There are, however, three control characters that must be escaped with a `^V`. They are `^T`, `^W`, and `^X`. So, for example, if you want to map `^T`, you must type:

    :map CTRL-V CTRL-T *sequence*

The use of CTRL-V applies to any `ex` command, not just a `map` command. This means that you can type a carriage return in an abbreviation or a substitution command. For example, the abbreviation:

    :**ab** 123 one^Mtwo^Mthree

expands to this:

    one
    two
    three

Here we show the sequence CTRL-V ENTER as `^M`, the way it would appear on your screen. (Vim highlights the `^M` in a different color so that you can tell it's actually a control character.)

You can also globally add lines at certain locations. The command:

    :**g**/^Section/s//As you recall, **in**^M&/

inserts, before all lines beginning with the word *Section*, a phrase on a separate line. The & restores the search pattern.

Unfortunately, one character always has special meaning in `ex` commands, even if you try to quote it with CTRL-V. Recall that the vertical bar (|) has special meaning as a separator of multiple `ex` commands. You cannot use a vertical bar in insert mode maps.

Now that you've seen how to use CTRL-V to protect certain keys inside `ex` commands, you're ready to define some powerful map sequences.

A Complex Mapping Example

Assume that you have a glossary with entries like this:

    map - an ex command which allows you to associate
    a complex command sequence with a single key.

You would like to convert this glossary list to a custom XML format, so that it looks like this:

```
<glossaryitem>
<name>map</name>
<para>An ex command...
```

The best way to define a complex map is to do the edit once manually, writing down each keystroke that you have to type, and then re-create these keystrokes as a map. You want to execute the following sequence:

1. Insert the `<glossaryitem>` tag, a newline, and the `<name>` tag.

2. Press ESC to terminate insert mode.

3. Move to the end of the first word (`e`) and add the `</name>` tag, a newline, and the `<para>` tag.

4. Press ESC to terminate insert mode.

5. Move forward one character, off of the closing `>` character (`l`).

6. Remove the space, the hyphen, and the following space (`3x`) and capitalize the next word (`~`).

That will be quite an editing chore if you have to repeat it more than just a few times.

With `:map` you can save the entire sequence so that it can be re-executed with a single keystroke:

```
:map g I<glossaryitem>^M<name>^[ea</name>^M<para>^[l3x~
```

Note that you have to "quote" both the ESC and the ENTER characters with CTRL-V. `^[` is the sequence that appears when you type CTRL-V followed by ESC. `^M` is the sequence shown when you type CTRL-V ENTER.

Now, simply typing `g` performs the entire series of edits. On a slow connection you can actually see the edits happening individually. On a fast one it will seem to happen by magic.

Don't be discouraged if your first attempt at key mapping fails. A small error in defining the map can give very different results from the ones you expect. Type `u` to undo the edit, and then try again. (That's exactly what we had to do while developing this map.)

For some command input maps that make editing XML easier, see the later section

More Examples of Mapping Keys

The following examples give you an idea of the clever shortcuts possible when defining keyboard maps:

1. Add text whenever you move to the end of a word:

       :map e ea

   Most of the time, the only reason you want to move to the end of a word is to add text. This map sequence puts you in insert mode automatically. Note that the mapped key, e, has meaning in vi. You're allowed to map a key that is already used by vi, but the key's normal function is unavailable as long as the map is in effect. This isn't so bad in this case, since the E command is often identical to e.

2. Transpose two words:

       :map K dwElp

   We discussed this sequence earlier in the chapter, but now you need to use E (assume here, and in the remaining examples, that the e command is mapped to ea). Remember that the cursor begins on the first of the two words. Unfortunately, because of the l command, this sequence (and the earlier version) doesn't work if the two words are at the end of a line: during the sequence, the cursor ends up at the end of the line, and l cannot move further right.

3. Save a file and edit the next one in a series:

       :map q :w^M:n^M

   (Use CTRL-V ENTER to get ^M into the map.) Notice that you can map keys to ex commands, but be sure to finish each ex command with a carriage return. This sequence makes it easy to move from one file to the next and is useful when you've opened many short files with one vi command. Mapping the letter q helps you remember that the sequence is similar to a "quit."[6]

4. Put troff emboldening codes around a word:

       :map v i\fB^[e\fP^[

   This sequence assumes that the cursor is at the beginning of the word. First you enter insert mode, and then you type the code for the bold font. In map commands, you don't need to type two backslashes to produce one backslash. Next, you return to command mode by typing a "quoted" ESC. Finally, you append the closing troff code at the end of the word, and you return to command mode.

---

6 Vim provides :wn to do this combined operation, but a :map q :wn^M would still be useful.

Notice that when we appended to the end of the word, we didn't need to use `ea`, since this sequence is itself mapped to the single letter `e`. This shows you that map sequences are allowed to contain other mapped commands. The ability to use nested map sequences is controlled by the `remap` option, which is normally enabled.

5. Put HTML emboldening codes around a word, even when the cursor is not at the beginning of the word:

       :map V lbi<B>^[e</B>^[

This sequence is similar the previous one; besides using HTML instead of `troff`, it uses `lb` to handle the additional task of positioning the cursor at the beginning of the word. The cursor might be in the middle of the word, so you want to move to the beginning with the `b` command. But if the cursor were already at the beginning of the word, the `b` command would move the cursor to the previous word instead. To guard against that, type an `l` before moving back with `b` so that the cursor never starts on the first letter of the word. You can define variations of this sequence by replacing the `b` with `B` and the `e` with `Ea`. In all cases, though, the `l` command prevents this sequence from working if the cursor is at the end of a line. (You could append a space to get around this.)

6. Repeatedly find and remove parentheses from around a word or phrase:[7]

       :map = xf)xn

This sequence assumes that you first found an open parenthesis by typing /(followed by ENTER.

If you choose to remove the parentheses, use the `map` command: delete the open parenthesis with `x`, find the closing one with `f)`, delete it with `x`, and then repeat your search for an open parenthesis with `n`.

If you don't want to remove the parentheses (for example, if they're being used correctly), don't use the mapped command: press `n` instead to find the next open parenthesis.

You could also modify the map sequence in this example to handle matching pairs of quotes.

7. Place C/C++ comments around an entire line:

       :map g I/* ^[A */^[

This sequence inserts `/*` at the line's beginning and appends `*/` at the line's end. You could also map a substitute command to do the same thing:

---

7 From "vi Tips for Power Users" by Walter Zintz, in *UnixWorld*, April 1990. This is intended for parenthesized text, not nested parenthesized equations.

```
:map g :s;.*;/* & */;^M
```

Here, you match the entire line (with `.*`), and when you replay it (with `&`), you surround the line with the comment symbols. Note the use of semicolon delimiters to avoid having to escape the `/` characters in the comment.

As a final word, you should know that there are many keys that either perform the same tasks as other keys or are rarely used. (For example, `^J` acts the same as `j`.) However, you should be familiar with the `vi` commands before you boldly disable their normal use by using them in map definitions.

Mapping Keys for Insert Mode

Normally, maps apply only to command mode—after all, in insert mode, keys stand for themselves and shouldn't be mapped as commands. However, by adding an exclamation mark (`!`) to the `map` command, you can force it to override the ordinary meaning of a key and produce the map in insert mode. This feature is useful when you find yourself in insert mode but need to escape briefly to command mode, run a command, and then return to insert mode.

For example, suppose you just typed a word but forgot to italicize it (or place quotes around it, etc.). You can define this map:

```
:map! + ^[bi<I>^[ea</I>
```

Now when you type a `+` at the end of a word, you surround the word with HTML italicization codes. The `+` won't show up in the text.

The sequence just shown escapes to command mode (`^[`), backs up to insert the first code (`bi<I>`), escapes again (`^[`), and moves ahead to append the second code (`ea</I>`). Since the map sequence begins and ends in insert mode, you can continue entering text after marking the word.

Here's another example: suppose that you've been typing your text and you realize that the previous line should have ended with a colon. You can correct that by defining this map sequence:[8]

```
:map! % ^[kA:^[jA
```

Now if you type a `%` anywhere along your current line, you'll append a colon to the end of the previous line. This command escapes to command mode, moves up a line, and appends the colon (`^[kA:`). The command then escapes again, moves down to the line you were on, and leaves you in insert mode at the end of the line (`^[jA`).

---

8 From "vi Tips for Power Users."

Note that we used uncommon characters (+ and %) for the previous map commands. When a character is mapped for insert mode, you can no longer type that character as text (unless you precede it with CTRL-V).

To reinstate a character for normal typing, use the command:

```
:unmap! x
```

where *x* is the character that was previously mapped for insert mode. (Although the editor expands *x* on the command line as you type it, making it look like you are unmapping the expanded text, it does correctly unmap the character.)

Insert-mode mapping is often more appropriate for tying character strings to special keys that you wouldn't otherwise use. It is especially useful with programmable function keys, as we're about to describe.

Mapping Function Keys

In the past, serial terminals came with programmable function keys. Using a special setup mode in the terminal, you would set up these keys to send whatever character or characters you wanted. Application programs could then take advantage of these function keys, using them as "shortcuts" for common or important actions.

Today's personal computer and laptop keyboards also have function keys, usually 10 or 12 keys in a row at the top, labeled F1 through F12 . Instead of setting up their behavior via a special hardware mode, what they do is defined by the terminal emulators and other programs running on your system.

Since the terminal emulators have entries in the `terminfo` database, the editor can recognize the escape sequences generated by the function keys, allowing you to map them to specific actions should you so choose. This is done in `ex`, using the syntax:

```
:map #1 commands
```

for function key number 1, and so on.

As with other keys, maps apply by default to command mode, but by using the `map!` command as well, you can define two separate values for a function key—one to be used in command mode, the other in insert mode. For example, if you are an HTML user, you might want to put font-switch codes on function keys, like so:[9]

```
:map #2 i<I>^[
:map! #2 <I>
```

If you are in command mode, the first function key enters insert mode, types in the three characters `<I>`, and returns to command mode. If you are already in insert

---

9 Function key F1 is often a "help" key, reserved for use by the terminal emulator; thus our example uses F2 .

mode, the key simply types the three-character HTML code. If the sequence contains ^M, which is a carriage return, press CTRL-V CTRL-M.

For instance, in order to have F2 available for mapping, the terminal database entry for your terminal must have a definition of k2, such as:

```
k2=^A@^M
```

In turn, the characters from the definition:

```
^A@^M
```

must be what is output when you press that key.

Seeing what function keys generate

To see what a function key puts out, use the od (octal dump) command with the -c option (show each character). You need to press ENTER after the function key, and then CTRL-D to get od to print the information. For example:

```
$ od -c              od reads from standard input
^[[A                 function key pressed
^D                   Control-D, EOF
0000000 033   [    [    A   \n
0000005
```

Here, the function key sent Escape, two left brackets, and an A.

Mapping Other Special Keys

Many keyboards have special keys, such as HOME, END, PAGE UP, and PAGE DOWN, that duplicate vi commands. If your terminal emulator's terminfo description is complete, the editor recognizes these keys. But if it isn't, you can use the map command to make them available. These keys generally send an escape sequence to the computer—an Escape character followed by a string of one or more other characters. To trap the Escape, you should press ^V before pressing the special key in the map. For example, to map the HOME key on a standard keyboard to a reasonable vi equivalent, you might define the following map:

```
:map CTRL-V HOME 1G
```

This appears on your screen as:

```
:map ^[[H 1G
```

Similar map commands display as follows:[10]

```
:map CTRL-V END G           displays    :map ^[[Y G
:map CTRL-V PAGE UP ^F       displays    :map ^[[V ^F
:map CTRL-V PAGE DOWN ^B      displays    :map ^[[U ^B
```

---

10 What you'd see on your system is likely different from the escape sequences shown here.

You'll probably want to place these maps in your *.exrc* file. Note that if a special key generates a long escape sequence (containing multiple nonprinting characters), ^V quotes only the initial escape character, and the map doesn't work. You will have to find the entire escape sequence (using od, as shown earlier) and type it in manually, quoting at the appropriate points, rather than simply pressing ^V and then the key.

If you use different kinds of terminals (such as both the command window on a Windows system and an xterm), you cannot expect that mappings like those just presented will always work. For this reason, Vim provides a portable way to describe such key mappings:

```
:map <Home> 1G          Enter six characters: < H o m e > (Vim)
```

vi and Vim typically provide these mappings as described, and if not, you can map them as just directed. However, we find such mappings to be counter to the vi philosophy of "never leave the keyboard." When we show users how to use Vim, one of the first things we do (or recommend) is to map the ⃞HOME⃞, ⃞END⃞, ⃞PAGE UP⃞, ⃞PAGE DOWN⃞, ⃞INSERT⃞, ⃞DELETE⃞, and all arrow keys to "no operation" to encourage learning native vi commands. The end result is more efficient editing and better muscle memory for the real vi commands, and a bunch of keys are now available to map for other heavy lifting.

For example, one of us regularly edits datafiles from four ice cream shops that he manages. The data has occasional lines with sales information, whitespace separated, for the stores for a day. He uses Vim's autocommand feature to detect that he's editing a file matching the shop names. (For more information on autocommands, see the section "Autocommands" on page 300.) This in turn defines the ⃞END⃞ key to sum the four values and to change the line to show the values *and* the total. The mapping looks like this:[11]

```
:noremap <end> !!awk 'NF == 4 && $1 + $2 + $3 + $4 > 0 {
  printf "\%s  total: $\%.2f\n", $0, $1 + $2 + $3 + $4;
  exit }; { print $0 }'<cr>
```

This is one long line in the *.vimrc* file; we split it across several lines so that it fits on the page. So the "heavy lift" is to use the ⃞END⃞ key on a line in the datafile that looks like:

```
450 235 1002 499
```

which gets converted to:

```
450 235 1002 499  total: $2186.00
```

---

11 The % sign has to be escaped in the mapping so that Vim doesn't replace it with the current filename.

when he presses the [END] key. Note that there is a double sanity check in the map, using an `awk` command to do the math, but only if there are four fields (`NF == 4`), and only if adding them results in a value greater than zero. (See `:help :map-modes` for information about the `:noremap` command.)

Mapping Multiple Input Keys

Mapping multiple keystrokes is not restricted just to function keys. You can also map sequences of regular keystrokes. This can help make it easier to enter certain kinds of text, such as DocBook XML or HTML.

Here are some `:map` commands, thanks to Jerry Peek, that make it easier to enter DocBook XML markup. The lines beginning with a double quote are comments (this is discussed later in the section "Comments in ex Scripts" on page 142):

```
:set noremap
" bold:
map! =b </emphasis>^[F<i<emphasis role="bold">
map =B i<emphasis role="bold">^[
map =b a</emphasis>^[
" Move to end of next tag:
map! =e ^[f>a
map =e f>
" footnote (tacks opening tag directly after cursor in text-input mode):
map! =f <footnote>^M<para>^M</para>^M</footnote>^[k0
" Italics ("emphasis"):
map! =i </emphasis>^[F<i<emphasis>
map =I i<emphasis>^[
map =i a</emphasis>^[
" paragraphs:
map! =p ^[jo<para>^M</para>^[0
map =P O<para>^[
map =p o</para>^[
" less-than:
map! *l &lt;
...
```

Using these commands, to enter a footnote you would enter insert mode and type `=f`. The editor would then insert the opening and closing tags and leave you in insert mode between them:

```
All the world's a stage.<footnote>
<para>█
</para>
</footnote>
```

These macros proved quite useful during the development of earlier editions of this book; they could be easily adapted to a different markup language, such as AsciiDoc, LaTeX, Texinfo, or Sphinx.

@-Functions

Named registers provide yet another way to create *macros*—complex command sequences that you can repeat with only a few keystrokes.

If you type a command line in your text (either a `vi` sequence or an `ex` command *preceded by a colon*) and then delete it into a named register, you can execute the contents of that register with the `@` command. For example, open a new line and enter:

```
cwgadfly CTRL-V ESC
```

This appears as:

```
cwgadfly^[
```

on your screen. Press `ESC` again to exit insert mode, and then delete the line into register g by typing `"gdd`. Now whenever you place the cursor at the beginning of a word and type `@g`, that word in your text is changed to *gadfly*.[12]

Since `@` is interpreted as a `vi` command, a dot (.) repeats the entire sequence, even if the register contains an `ex` command. `@@` repeats the last `@`, and `u` or `U` can be used to undo the effect of `@`.

Vim makes it easier to save text into a named register. In `vi` command mode, a `q` followed by a register name starts recording what you type into the named register. Use `q` by itself to end recording. Vim puts a message that it's recording into the status line to remind you. Using register `a` for the previous example in Vim, you would type `qacwgadfly^[q`, which you could then execute with `@a`.

This is a simple example. @-functions are useful because they can be adapted to very specific commands. They are especially useful when you are editing between files, because you can store the commands in their named registers and access them from any file you edit. @-functions are also useful in combination with the global replacement commands discussed in Chapter 6, "Global Replacement".

Of course, if you use named registers both for @-functions and for storing yanked or deleted text, you want to be careful to keep them separate, perhaps using letters earlier in the alphabet for storage and reserving letters toward the end of the alphabet for @-functions.

---

12 This is a little tricky. The `dd` gets the newline at the end of the line too, causing `@g` to move the cursor down one line after changing the current word. To do this exactly right, you have to type `"gdf^V^[`. Whew!

Executing Registers from ex

You can also execute text saved in a register from `ex` mode. In this case, you would enter an `ex` command, delete it into a named register, and then use the `@` command from the `ex` colon prompt. For example, enter the following text:

```
ORA publishes great books.
ORA is my favorite publisher.
1,$s/ORA/O'Reilly Media/g
```

With your cursor on the last line, delete the command into the `g` register: `"gdd`. Move your cursor to the first line: `kk`. Then execute the register from the colon command line: `:@g` `ENTER`. Your screen should now look like this:

```
O'Reilly Media publishes great books.
O'Reilly Media is my favorite publisher.
```

Some versions of `vi` treat `*` identically to `@` when used from the `ex` command line. Vim also does this, but only if the `compatible` option is set. In addition, if the register character supplied after the `@` or `*` command is `*`, the command is taken from the default (unnamed) register.

Using ex Scripts

Certain `ex` commands you use only within `vi`, such as maps, abbreviations, and so on. If you store these commands in your *.exrc* file, the commands are automatically executed when you invoke `vi` or Vim. Any file that contains commands to execute is called a *script*.

The commands in a typical *.exrc* script are of no use outside `vi`. However, you can save other `ex` commands in a script and then execute the script on a file or on multiple files. Mostly you'll use substitute commands in these external scripts.

For a (technical) writer, a useful application of `ex` scripts is to ensure consistency of terminology—or even of spelling—across a document set. For example, let's assume that you've run the Unix `spell` command on two files and that the command has printed out the following list of misspellings:

```
$ spell sect1 sect2
chmod
ditroff
myfile
thier
writeable
```

As is often the case, `spell` has flagged a few technical terms and special cases it doesn't recognize, but it has also identified two genuine spelling errors.

Because we checked two files at once, we don't know which files the errors occurred in or where they are in the files. Although there are ways to find this out, and the

job wouldn't be too hard for only two errors in two files, you can easily imagine how time-consuming the job could grow to be for a poor speller or for a typist proofing many files at once.

To make the job easier, you could write an `ex` script containing the following commands:

```
%s/thier/their/g
%s/writeable/writable/g
wq
```

Assume you've saved these lines in a file named *exscript*. The script could be executed from within `vi` with the command:

```
:so exscript
```

or the script can be applied to a file right from the command line. Then you could edit the files *sect1* and *sect2* as follows:

```
$ ex -s sect1 < exscript
$ ex -s sect2 < exscript
```

The `-s` (for either "script mode" or "silent mode") following the invocation of `ex` is the POSIX way to tell the editor to suppress the normal terminal messages.[13]

If the script were longer than the one in our simple example, we would already have saved a fair amount of time. However, you might wonder if there isn't some way to avoid repeating the process for each file to be edited. Sure enough, we can write a shell script that includes—but generalizes—the invocation of `ex`, so that it can be used on any number of files.

Looping in a Shell Script

You may know that the shell is a programming language as well as a command-line interpreter. To invoke `ex` on a number of files, we use a simple type of shell script command called the `for` loop. A `for` loop allows you to apply a sequence of commands for each argument given to the script. The `for` loop is probably the single most useful piece of shell programming for beginners. You'll want to remember it even if you don't write any other shell programs.

Here's the syntax of a `for` loop:

```
for variable in list
do
    command(s)
done
```

---

13 Traditionally, ex used a single minus sign for this purpose. Typically, for backward compatibility, both are accepted.

For example:

```
for file in "$@"
do
    ex -s "$file" < exscript
done
```

(The ex command doesn't need to be indented; we indented it for clarity.) Quoting the expansion of the file variable ($file) allows the script to work even when filenames have spaces in their names.[14]

After we create this shell script, we save it in a file called *correct* and make it executable with the command chmod 755 correct. Now type the following:

```
$ ./correct sect1 sect2
```

The for loop in *correct* assigns each argument (each file in the list specified by "$@", which stands for *all arguments*) to the variable file and executes the ex script on the contents of that variable.

It may be easier to grasp how the for loop works with an example whose output is more visible. Let's look at a script to rename files:

```
for file in "$@"
do
    mv "$file" "$file.x"
done
```

Assuming this script is in an executable file called *move*, here's what we can do:

```
$ ls
ch01 ch02 ch03 move
$ ./move ch??              Just the chapter files
$ ls                       Check the results
ch01.x ch02.x ch03.x move
```

With creativity, you could rewrite the script to rename the files more specifically:

```
for nn in "$@"
do
    mv "ch$nn" "sect$nn"
done
```

With the script written this way, you'd specify numbers instead of filenames on the command line:

```
$ ls
ch01 ch02 ch03 move
$ ./move 01 02 03
$ ls
sect01 sect02 sect03 move
```

---

14 Spaces in filenames are not good practice but are also not uncommon. Robust scripts should also work for such files.

The `for` loop need not take `"$@"` (all arguments) as the list of values to be substituted. You can specify an explicit list as well. For example:

```
for variable in a b c d
```

assigns *variable* to *a*, *b*, *c*, and *d* in turn. Or you can substitute the output of a command. For example:

```
for variable in $(grep -l "Alcuin" *)
```

assigns *variable* in turn to the name of each file in which `grep` finds the string *Alcuin*. (`grep -l` prints the filenames whose contents match the pattern, without printing the actual matching lines.)

If no list is specified:

```
for variable
```

the variable is assigned to each command-line argument in turn, much as it was in our initial example. The four-character sequence `"$@"` expands to `"$1"`, `"$2"`, `"$3"`, and so on. Quotation marks prevent further interpretation of special characters and keep filenames with spaces in their names as single items.

Let's return to our main point and our original script:

```
for file in "$@"
do
    ex -s "$file" < exscript
done
```

It may seem a little inelegant to have to use two scripts—the shell script and the `ex` script. And in fact, the shell does provide a way to include an editing script inside a shell script, as we are about to see.

Here Documents

In a shell script, the operator `<<` means to take the following lines, up to a specified string, as input to a command. (This is often called a *here document*.) Using this syntax, we could include our editing commands in *correct* like this:

```
for file in "$@"
do

ex -s "$file" << end-of-script
g/thier/s//their/g
g/writeable/s//writable/g
wq
end-of-script

done
```

The string `end-of-script` is entirely arbitrary—it just needs to be a string that won't otherwise appear in the input and can be used by the shell to recognize when the here

document is finished. It also *must* be placed at the start of the line. By convention, many users specify the end of a here document with the string `EOF`, or `E_O_F`, to indicate the end of the file.

There are advantages and disadvantages to each approach shown. If you want to make a one-time series of edits and don't mind rewriting the script each time, the here document provides an effective way to do the job.

However, it's more flexible to write the editing commands in a separate file from the shell script. For example, you could establish the convention that you always put editing commands in a file called *exscript*. Then you only need to write the *correct* script once. You can store it away in your personal "tools" directory (which you've added to your search path) and use it whenever you like.

Sorting Text Blocks: A Sample ex Script

Suppose you want to alphabetize a file of glossary definitions encoded in a custom version of XML. Each definition is bracketed by `<glossaryitem>` and `</glossary item>`. The name of each item is bracketed by `<name>` and `</name>`. The glossary file looks something like this:

```
<glossaryitem>
<name>TTY_ARGV</name>
<para>The command, specified as an argument vector,
that the TTY subwindow executes.</para>
</glossaryitem>
<glossaryitem>
<name>ICON_IMAGE</name>
<para>Sets or gets the remote image for icon's image.</para>
</glossaryitem>
<glossaryitem>
<name>XV_LABEL</name>
<para>Specifies a frame's header or an icon's label.</para>
</glossaryitem>
<glossaryitem>
<name>SERVER_SYNC</name>
<para>Synchronizes with the server once.
Does not set synchronous mode.</para>
</glossaryitem>
```

You can alphabetize a file by running the lines through the Unix `sort` command, but you don't really want to sort on a line-by-line basis. You want to sort only the glossary terms, moving each definition—untouched—along with its corresponding term. As it turns out, you can treat each text block as a unit by joining the block into one line. Here's the first version of your `ex` script:

```
g/^<glossaryitem>/,/^<\/glossaryitem>/j
%!sort
wq
```

Each glossary entry is found between `<glossaryitem>` and `</glossaryitem>` tags. (Note the use of `\/` to escape the slash in the closing tag.) `j` is the `ex` command to join a line (the equivalent in `vi` mode is `J`). So the first command joins every glossary entry into one "line." The second command then sorts the file, producing lines like this:

```
<glossaryitem> <name>ICON_IMAGE</name> <para>Sets ... </glossaryitem>
<glossaryitem> <name>SERVER_SYNC</name> <para>Synchronizes ... </glossaryitem>
<glossaryitem> <name>TTY_ARGV</name> <para>The command, ... </glossaryitem>
<glossaryitem> <name>XV_LABEL</name> <para>Specifies ... </glossaryitem>
```

The lines are now sorted by glossary entry; unfortunately, each line also has XML tags and text mixed in (we've used ellipses […] to show omitted text). Somehow, you need to insert newlines to "un-join" the lines. You can do this by modifying your `ex` script: mark the joining points of the text blocks *before* you join them, and then replace the markers with newlines. Here's the expanded `ex` script:

```
g/^<glossaryitem>/,/^<\/glossaryitem>/-1s/$/@@/    Append @@ to the end of each line
g/^<glossaryitem>/,/^<\/glossaryitem>/j            Join the entries
%!sort                                             Sort them
%s/@@ /^@/g                                        Break the lines apart
wq                                                 Save the file
```

The first three commands produce lines like this:

```
<glossaryitem>@@ <name>ICON_IMAGE</name>@@ <para>Sets ...</para>@@ </glossaryitem>
<glossaryitem>@@ <name>SERVER_SYNC</name>@@ <para>Synchronizes ...</para>@@ </glossaryitem>
<glossaryitem>@@ <name>TTY_ARGV</name>@@ <para>The command, ...</para>@@ </glossaryitem>
<glossaryitem>@@ <name>XV_LABEL</name>@@ <para>Specifies ...</para>@@ </glossaryitem>
```

Note the extra space following each `@@`. The spaces result from the `j` command, because it converts each newline into a space.

The first command marks the original line breaks with `@@`. You don't need to mark the end of the block (after the `</glossaryitem>`), so the first command uses a `-1` to move back up one line at the end of each block. The fourth command restores the line breaks by replacing the markers (plus the extra space) with newlines. (You enter the newline as CTRL-V CTRL-J. This is discussed shortly.) Now your file is sorted by blocks.

A subtle vi/Vim difference

In the script we just finished writing for use with Vim's version of `ex`, we entered a newline as CTRL-V CTRL-J, and it showed up as `^@`. When editing interactively in Vim, however, you must do this differently; you enter the newline as CTRL-V ENTER, and it shows up as `^M`.

The original `vi` doesn't distinguish. If you're using the original `vi`, you enter the newline, both interactively and in a script, as CTRL-V ENTER.

Comments in ex Scripts

You may want to reuse a script like the one we just developed, adapting it to a new situation. With a complex script like this, it is wise to add comments so that it's easier for someone else (or even yourself!) to reconstruct how it works. In ex scripts, anything following a double quote is ignored during execution, so a double quote can mark the beginning of a comment. Comments can go on their own line. They can also go at the end of any command that doesn't interpret a quote as part of the command. For example, a quote has meaning to map commands and shell escapes, so you can't end such lines with a comment.

Besides using comments, you can specify a command by its full name, something that would ordinarily be too time-consuming from within vi. Finally, if you add spaces, the ex script shown previously becomes this more readable one:

```
" Mark lines between each <glossaryitem>...</glossaryitem> block
global /^<glossaryitem>/,/^\/glossaryitem>/-1 substitute /$/@@/
" Now join the blocks into one line
global /^<glossaryitem>/,/^\/glossaryitem>/ join
" Sort each block--now really one line each
%!sort
" Restore the joined lines to original blocks
%substitute /@@ /^@/g
" Write the file back out and exit
wq
```

> In previous editions of this book, we wrote:
>
> > Surprisingly, the substitute command does not work in ex, even though the full names for the other commands do.
>
> And that was correct at the time, at least for the Solaris version of vi.
>
> However, upon testing the "Heirloom" and Solaris 11 versions of vi, we found that substitute worked just fine as an ex command. Nonetheless, before using the full command in a script, you should check out your local version and make sure that it works.

Beyond ex

If this discussion has whetted your appetite for even more editing power, you should be aware that Unix systems provide editors even more powerful than ex: the sed stream editor and the awk data manipulation language. There is also the extremely popular perl programming language. For information on these programs, see the O'Reilly books *sed & awk* by Dale Dougherty and Arnold Robbins, *Effective awk Programming* by Arnold Robbins, *Learning Perl* by Randal L. Schwarz, brian d foy,

and Tom Phoenix, and *Programming Perl* by Tom Christiansen, brian d foy, Larry Wall, and Jon Orwant.

Editing Program Source Code

All of the features discussed so far are of interest whether you are editing regular text or program source code. However, there are a number of additional features that are of interest chiefly to programmers. These include indentation control, searching for the beginning and end of procedures, and using `ctags`.

The following discussion is adapted from documentation provided by MKS, Inc. (formerly Mortice Kern Systems), with its excellent implementation of `vi` for MS-DOS- and Windows-based systems, available as a part of the MKS Toolkit. It is reprinted by permission of MKS, Inc.

Indentation Control

Source code for a program differs from ordinary text in a number of ways. One of the most important of these is the way in which source code uses indentation. Indentation shows the logical structure of the program: the way in which statements are grouped into blocks. `vi` provides automatic indentation control. To use it, issue the command:

```
:set autoindent
```

Now when you indent a line with spaces or tabs, the following lines are automatically indented by the same amount. When you press ENTER after typing the first indented line, the cursor goes to the next line and automatically indents the same distance as the previous line.

As a programmer, you will find this saves you quite a bit of work getting the indentation right, especially when you have several levels of indentation.

When you are entering code with autoindent enabled, typing CTRL-T at the start of a line gives you another level of indentation, and typing CTRL-D takes one level away.

We should point out that CTRL-T and CTRL-D are typed while you are in insert mode, unlike most other commands, which are typed in command mode.

There are two additional variants of the CTRL-D command:

`^ ^D`

When you type `^ ^D` (^ CTRL-D), the editor shifts the cursor back to the beginning of the line, but only for the current line. The next line you enter will start at the current autoindent level. This is particularly useful for entering C preprocessor commands while typing in C/C++ source code.

```
0 ^D
```

When you type `0  ^D`, the editor shifts the cursor back to the beginning of the line. In addition, the current autoindent level is reset to zero; the next line you enter is not autoindented.

Finally, there is your terminal's *line erase* character, typically CTRL-U, which erases the entire input line you've typed so far. This also works in the GUI version of Vim.

Try using the `autoindent` option when you are entering source code. It simplifies the job of getting indentation correct. It can even sometimes help you avoid bugs—e.g., in C source code, where you usually need one closing curly brace (}) for every level of indentation you go backward.

The << and >> commands are also helpful when indenting source code. By default, >> shifts a line right eight spaces (i.e., adds eight spaces of indentation) and << shifts a line left eight spaces. For example, move the cursor to the beginning of a line and press > twice (>>). You will see the line move right. If you now press < twice (<<), the line moves back again.

You can shift a number of lines by typing the number followed by >> or <<. For example, move the cursor to the first line of a good-sized paragraph and type 5>>. This shifts the first five lines in the paragraph.

The default shift is eight spaces (right or left). This default can be changed with a command such as:

```
:set shiftwidth=4
```

It is convenient to have a `shiftwidth` that is the same size as the width between tab stops. The default tab stop is also eight character positions.

The editor attempts to be smart when doing indenting. Usually, when you see text indented by eight spaces at a time, it actually inserts tab characters into the file, since tabs usually expand to eight spaces. This is the Unix default; it is most noticeable when you type a tab during normal input and when files are sent to a printer—Unix expands them with a tab stop of eight spaces.

If you wish, you can change how tabs are represented on your screen by changing the `tabstop` option. For example, if you have something that is deeply indented, you might wish to use a tab stop setting of every four characters, so that the lines do not wrap. The following command makes this change:

```
:set tabstop=4
```

You should change the `shiftwidth` at the same time and to the same value as the tab stop.

Change your tab stops with consideration. Although `vi` and Vim can display the file using an arbitrary tab stop setting, the tab characters in your files are still expanded using an eight-character tab stop by many other Unix programs.

Even worse: mixing tabs, spaces, and unusual tab stops makes your file completely unreadable when viewed outside the editor, with a pager such as `more`, or when printed.

When programming and dealing with tabs and tab stops, you have two alternatives:

- Accept that eight-character tab stops are a fact of life on Unix, and just get used to them.

- Have the editor expand tabs into spaces as you enter them. You do this with:

    `:set expandtab`

    When `expandtab` is set, every time you hit the [TAB] key, the editor enters enough spaces to move the cursor over to the next tab stop.

    If everyone on your team does this, then all your code will be formatted consistently, and things will work well. This is particularly important for a language like Python, where indentation of your code indicates statement grouping and mismatched spaces and tabs becomes a recipe for disaster.[15] As a side note, you can use the `expand` utility to convert preexisting tabs to spaces.

Sometimes indentation won't work the way you expect, because what you believe to be a tab character is actually one or more spaces. Normally, your screen displays both a tab and a space as whitespace, making the two indistinguishable. You can, however, issue the command:

`:set list`

This alters your display so that a tab appears as the control character `^I` and an end-of-line appears as a `$`. This way, you can spot a true space, and you can see extra spaces at the end of a line. A temporary equivalent is the `:l` command. For example, the command:

`:5,20 l`

displays lines 5 through 20, showing tab characters and end-of-line characters.

---

15 Spaces versus tabs can be a religious issue. See this wonderful segment from the *Silicon Valley* TV show: *https://www.youtube.com/watch?v=SsoOG6ZeyUI*.

A Special Search Command

The characters (, [, and { can all be called opening brackets. When the cursor is resting on one of these characters, pressing the % key moves the cursor from the opening bracket forward to the corresponding closing bracket—),], or }—keeping in mind the usual rules for nesting brackets.[16] For example, if you were to move the cursor to the first (in:

```
if ( cos(a[i]) == sin(b[i]+c[i]) )
{
    printf("cos and sin equal!\n");
}
```

and press %, you would see that the cursor jumps to the parenthesis at the end of the line. This is the closing parenthesis that matches the opening one.

Similarly, if the cursor is on one of the closing bracket characters, pressing % moves the cursor backward to the corresponding opening bracket character. For example, move the cursor to the closing brace after the `printf` line just shown and press %.

The editor is even smart enough to find a bracket character for you. If the cursor is not on a bracket character, when you press %, it searches forward on the current line to the first open or close bracket character it finds, and then it moves to the matching bracket! For instance, with the cursor on the = in the first line of the example just shown, % finds the open parenthesis and then moves to the close parenthesis.

Not only does this search character help you move forward and backward through a program in long jumps, it also lets you check the nesting of brackets and parentheses in source code. For example, if you put the cursor on the first { at the beginning of a C function, pressing % should move you to the } that (you think) ends the function. If it's the wrong one, something has gone wrong somewhere. If there is no matching } in the file, the editor beeps at you.[17]

Another technique for finding matching brackets is to turn on the following option:

```
:set showmatch
```

Unlike %, setting `showmatch` (or its abbreviation `sm`) helps you while you're in insert mode. When you type a) or a },[18] the cursor moves briefly back to the matching (or { before returning to your current position. If the match doesn't exist, the terminal beeps. If the match is merely off-screen, the editor silently keeps going. Using the `matchparen` plug-in, which is loaded by default, Vim can highlight the matching parenthesis or brace.

---

16 Some versions also match < and > with %.

17 Note that the editor also counts brackets inside quoted strings and comments, so % isn't foolproof.

18 In Vim, `showmatch` also shows you matching square brackets ([and]).

Using Tags

The source code for a large C or C++ program is usually spread over several files. Sometimes it is difficult to keep track of which file contains which function definitions. To simplify matters, you can use a Unix command called `ctags` together with the `:tag` command of `ex`.

You issue the `ctags` command at the shell command line. Its purpose is to create an information file that the editor can use later to determine which files define which functions. By default, this file is called *tags*. From within an editing session, a command of the form:

```
:!ctags file.c
```

creates a file named *tags* in your current directory that contains information on the functions defined in *file.c*. A command such as:

```
:!ctags *.c
```

creates a *tags* file describing all the C source files in the directory.

> Legacy Unix versions of `ctags` handle the C language and often Pascal and Fortran 77. Sometimes they even handle assembly language. Almost universally, however, they do not handle C++. Other versions are available that can generate *tags* files for C++ and for other languages and file types. For more information, see the section "Enhanced Tags" on page 148.

Now suppose your *tags* file contains information on all the source files that make up a C program. Also suppose that you want to look at or edit a function in the program, but you do not know where the function is. From within `vi` mode, the command:

```
:tag name
```

looks at the *tags* file to find out which file contains the definition of the function *name*. It then reads in that file and positions the cursor on the line where the name is defined. In this way, you don't have to know which file you have to edit; you only have to decide which function you want to edit.

You can use the tag facility from `vi`'s command mode as well. Place the cursor on the identifier you wish to look up, and then type CTRL-]. The editor performs the tag lookup and moves to the file that defines the identifier. Be careful where you place the cursor; the editor uses the "word" under the cursor starting at the current cursor position, not the entire word containing the cursor.

If you try to use the `:tag` command to read in a new file and you haven't saved your current text since the last time you changed it, the editor does not let you go to the new file. You must either write out your current file with the `:w` command and then issue `:tag`, or else type:

```
:tag! name
```

to override the editor's reluctance to discard edits.

Enhanced Tags

Unix `ctags` works but is limited. The "Exuberant `ctags`" program, written by Darren Hiebert, is a `ctags` clone that is considerably more capable than Unix `ctags`. It produced an extended *tags* file format that makes the tag searching and matching process more flexible and powerful.

Unfortunately, Exuberant `ctags` was last updated in 2009. The "Universal `ctags`" project provides a maintained version of `ctags`, starting from the Exuberant `ctags` code base.

In this section we first describe the Universal `ctags` program and then describe the enhanced tags file format.

This section also describes tag stacks: the ability to save multiple locations visited with the `:tag` or `^]` commands. Vim and (surprisingly enough) the Solaris versions of `vi` support tag stacking.

Universal ctags

The Universal `ctags` home page is at *https://ctags.io/*. The source code is at *https://github.com/universal-ctags/ctags*. You will have to do an internet search to see if your system's package manager allows you to install a precompiled version, or if you may need to build it from source code.

The following list of the program's features is adapted from the *old-docs/README.exuberant* file in the Universal `ctags` distribution:

- It is capable of generating tags for *all* types of C and C++ language tags, including class names, macro definitions, enum names, enumerators (values inside an enumeration), function (method) definitions, function (method) prototypes/declarations, structure members and class data members, struct names, typedefs, union names, and variables. (Whew!)
- It supports both C and C++ code.
- 41 languages are supported, including C# and Java.

- It is very robust in parsing code and is far less easily fooled by code containing `#if` preprocessor conditional constructs.

- It can be used to print out a human-readable list of selected objects found in source files.

- It supports generation of GNU Emacs–style *tags* files (*etags*).

- It works on a large variety of operating systems, including Unix, OpenVMS, and MS-Windows.

Universal `ctags` produces *tags* files in the form described next.

The new tags format

Traditionally, a *tags* file has three tab-separated fields: the tag name (typically an identifier), the source file containing the tag, and an indication of where to find the identifier. This indication is either a simple line number or a `nomagic` search pattern enclosed either in slashes or question marks. Furthermore, the *tags* file is always sorted.

This is the format generated by the Unix `ctags` program. In fact, many versions of `vi` allowed *any* command in the search pattern field (a rather gaping security hole). Furthermore, due to an undocumented implementation quirk, if the line ended with a semicolon and then a double quote (`;"`), anything following those two characters would be ignored. (The double quote starts a comment, as it does in *.exrc* files.)

The new format is backward compatible with the traditional one. The first three fields are the same: tag, filename, and search pattern. Universal `ctags` generates only search patterns, not arbitrary commands. Special attributes are placed after a separating `;"`. Each attribute is separated from the next by a tab character and consists of two colon-separated subfields. The first subfield is a keyword describing the attribute; the second is the actual value. Table 7-1 lists the supported keywords.

Table 7-1. Extended ctags keywords

Keyword	Meaning
`arity`	For functions. Defines the number of arguments.
`class`	For C++ member functions and variables. The value is the name of the class.
`enum`	For values in an `enum` data type. The value is the name of the `enum` type.
`file`	For tags that are "static," i.e., local to the file. The value should be the name of the file. If the value is given as an empty string (just `file:`), it is understood to be the same as the filename field; this special case was added partly for the sake of compactness, and partly to provide an easy way to handle *tags* files that aren't in the current directory. The value of the filename field is always relative to the directory in which the *tags* file itself resides.
`function`	For local tags. The value is the name of function in which they're defined.

Keyword	Meaning
kind	The value is a single letter that indicates the tag's lexical type. It can be f for a function, v for a variable, and so on. Since the default attribute name is kind, a solitary letter can denote the tag's type (e.g., f for a function).
scope	Intended mostly for C++ class member functions. It is usually private for private members or omitted for public members, so users can restrict tag searches to only public members.
struct	For fields in a struct. The value is the name of the structure.
union	For fields in a union. The value is the name of the union.

If the field does not contain a colon, it is assumed to be of type kind. Here are some examples:

```
ALREADY_MALLOCED  awk.h  /^#define ALREADY_MALLOCED /;" d
ARRAYMAXED        awk.h  /^  ARRAYMAXED = 0x4000,;"    e  enum:exp_node::flagvals
array.c  array.c  1;" F
```

ALREADY_MALLOCED is a C macro. ARRAYMAXED is a C enum defined in *awk.h*. The third line is a bit different: it is a tag for the actual source file! This is generated with the --extras=f option to Universal ctags, and it allows you to give the command :tag array.c. More usefully, you can put the cursor over a filename and use the ^] command to go to that file (for example, if you're editing a *Makefile* and wish to go to a particular source file).

Within the value part of each attribute, the backslash, tab, carriage return, and newline characters are encoded as \\, \t, \r, and \n, respectively.

Universal *tags* files may have some number of initial tags that begin with !_TAG_. These tags usually sort to the front of the file and are useful for identifying which program created the file. Here is what Universal ctags generates:

```
!_TAG_FILE_FORMAT      2       /extended format; --format=1 will not append ;" to lines/
!_TAG_FILE_SORTED      1       /0=unsorted, 1=sorted, 2=foldcase/
!_TAG_OUTPUT_EXCMD     mixed   /number, pattern, mixed, or combineV2/
!_TAG_OUTPUT_FILESEP   slash   /slash or backslash/
!_TAG_OUTPUT_MODE      u-ctags /u-ctags or e-ctags/
!_TAG_PATTERN_LENGTH_LIMIT    96      /0 for no limit/
!_TAG_PROC_CWD  /home/arnold/Gnu/gawk/gawk.git/ //
!_TAG_PROGRAM_AUTHOR   Universal Ctags Team    //
!_TAG_PROGRAM_NAME     Universal Ctags /Derived from Exuberant Ctags/
!_TAG_PROGRAM_URL      https://ctags.io/       /official site/
!_TAG_PROGRAM_VERSION  5.9.0   /p5.9.20201206.0/
```

Editors can take advantage of these special tags to implement special features. For example, Vim pays attention to the !_TAG_FILE_SORTED tag and uses a binary search to search the *tags* file instead of a linear search if the file is indeed sorted.

If you use *tags* files, we recommend that you get and install Universal ctags.

Tag stacks

The `ex` command `:tag` and the `vi` command-mode `^]` command provide a limited means of finding identifiers, based on the information provided in a *tags* file. Vim and the Solaris `vi` extend this ability by maintaining a *stack* of tag locations. Each time you issue the `ex` command `:tag` or use the `vi` mode `^]` command, the editor saves the current location before searching for the specified tag. You may then return to a saved location using (usually) the `vi` command `^T` or an `ex` command.

Solaris `vi` tag stacking and an example are presented next. Vim's tag stacking is described in the section .

Solaris vi

Surprisingly enough, the Solaris versions of `vi` support tag stacking. Perhaps not so surprisingly, this feature is completely undocumented in the Solaris *ex*(1) and *vi*(1) manual pages. For completeness, we summarize Solaris `vi` tag stacking in Tables 7-2, 7-3, and 7-4. Tag stacking in Solaris `vi` is quite simple.[19]

Table 7-2. Solaris `vi` tag commands: `ex` commands

Command	Function
`ta[g][!]` *tag string*	Edit the file containing *tagstring* as defined in the *tags* file. The `!` forces `vi` to switch to the new file if the current buffer has been modified but not saved.
`po[p][!]`	Pop the tag stack by one element.

Table 7-3. Solaris `vi` command mode tag commands

Command	Function
`^]`	Look up the location of the identifier under the cursor in the *tags* file, and move to that location. If tag stacking is enabled, the current location is automatically pushed onto the tag stack.
`^T`	Return to the previous location in the tag stack, i.e., pop off one element.

Table 7-4. Solaris `vi` options for tag management

Option	Function
`taglength, tl`	Controls the number of significant characters in a tag that is to be looked up. The default value of zero indicates that all characters are significant.
`tags`	The value is a list of filenames in which to look for tags. The default value is `"tags /usr/lib/tags"`.
`tagstack`	When set to true, `vi` stacks each location on the tag stack. Use `:set notagstack` to disable tag stacking.

---

19 This information was discovered based on experimentation. YMMV (your mileage may vary).

Universal ctags and Vim

To give you a feel for using tag stacks, we present a short example that uses Universal ctags and Vim.

Suppose you are working with a program that uses the GNU getopt_long function, and you need to understand more about it.

GNU getopt consists of three files: *getopt.h*, *getopt.c*, and *getopt1.c*.

First, you create the *tags* file, and then you start by editing the main program, found in *main.c*:

```
$ ctags *.[ch]
$ ls
getopt1.c getopt.c getopt.h getopt_int.h main.c Makefile tags
$ vim main.c
```

Keystrokes	Results
/getopt_	```/* option processing. ready, set, go! */
while ((c = getopt_long(argc, argv, optlist, optab, NULL)) != EOF) {
switch (c) {
case 'l':```
	Edit *main.c* and move to the call to getopt_long.
^]	```int getopt_long (int argc, char *const *argv, const char *options,
const struct option *long_options, int *opt_index)
{
return _getopt_internal (argc, argv, options, long_options, opt_index, 0);
}```
	"getopt1.c" 195L, 4750B
	Do a tag lookup on getopt_long. Vim moves to *getopt1.c,* placing the cursor on the definition of *getopt_long.*

It turns out that getopt_long is a "wrapper" function for _getopt_internal. You place the cursor on _getopt_internal and do another tag search:

Keystrokes	Results
3jf_ ^]	```int _getopt_internal (int argc, char *const *argv, const char *optstring,
                  const struct option *longopts, int *longind, int long_only)
{
  int result;

  getopt_data.optind = optind;
  getopt_data.opterr = opterr;

  result = _getopt_internal_r (argc, argv, optstring, longopts,
                                longind, long_only, &getopt_data);

  optind = getopt_data.optind;``` |
| | "getopt.c" 1294L, 34298B |

Keystrokes	Results
	You have now moved to *getopt.c*. To find out more about `struct option`, move the cursor to `option` and do another tag search.
| `jfo; ^]` | ```
    one).  For long options that have a zero `flag' field, `getopt'
    returns the contents of the `val' field.  */
struct option
{
    const char *name;
    /* has_arg can't be an enum because some compilers complain about
       type mismatches in all the code that assumes it is an int.  */
    int has_arg;
    int *flag;
    int val;
};

/* Names for the values of the `has_arg' field of `struct option'.  */

"getopt.h" 191L, 6644B
``` |
| | The editor moves to the definition of `struct option` in *getopt.h*. You may now look over the comments that explain how it's used. |
| `:tags` | ```
# TO tag         FROM line  in file/text
1  1 getopt_long          29  main.c
2  1 _getopt_internal     70  getopt1.c
3  1 option             1185  getopt.c
``` |
| | The `:tags` command in Vim displays the tag stack. |

Typing `^T` three times would move you back to *main.c*, where you started. The tag facilities make it easy to move around as you edit source code.

Vim

Part II describes the most popular `vi` clone, named Vim (which stands for "`vi` improved"). This part contains the following chapters:

Vim (vi Improved): Overview and Improvements over vi

"Look! Up in the sky! It's a bird!"
"It's a plane!"
"It's Superman!"

Yes, it's Superman! Strange visitor from another planet who came to Earth with powers and abilities far beyond those of mortal men.

　　　—The 1950s *Superman* TV Show

While Vim is neither strange nor from another planet, it *does* have powers and abilities far beyond those of mortal text editors!

In this chapter, we introduce the most noteworthy of Vim's many technical advances over vi, along with a bit of history. We continue with some pointers to special Vim modes and teaching tools for new users. We then move on to introduce some of Vim's improvements over vi, ranging from multiple color syntax definitions to full-blown scripting.

If vi is excellent (it is), Vim is amazing. In this first chapter of Part II we discuss how Vim fills in many features that users have complained were missing from vi. This chapter introduces:

- Editing enhancements over vi
- Built-in help
- Startup and initialization options
- New motion commands
- Extended regular expressions

- Extended undo
- Incremental searching
- Left-to-right scrolling

The remaining Part II chapters cover:

- The Vim graphical user interface (GUI)
- Multiwindow editing
- Programming enhancements
- Vim scripts
- Other cool stuff
- Some Vim power techniques

About Vim

Vim stands for "vi improved."[1] It was written and is maintained by Bram Moolenaar. Today, Vim is perhaps the most widely used vi implementation. As of this writing, the current version is 8.2.

Computer capability increased dramatically in the time between the seventh edition of this book and this edition. In 2008, 1 gigabyte was a lot of memory. And while memory became increasingly affordable and more expansive (i.e., more gigabytes), users still had to configure their applications and tools to share the available computing resources.

Today it's common to see 16 gigabytes of memory, and many computers ship with super fast drives, typically solid state drives (SSDs). These technology improvements obviate many old habits. Consequently, many Vim configuration suggestions provide greater upper limits for greater editing power. Later on, we discuss things like command and search history, as well as the history of changes for undos.

Unconstrained by standards or committees, Vim continues to grow in functionality. An entire community has grown up around it. Collectively, the members of the community decide what new features to add and what existing features to modify by nominating and voting for suggestions during development cycles.

Inspired by Bram's dedicated energy and the voting system, Vim enjoys a strong following. It maintains its value by growing and changing with the computing industry

---

[1] We note that Wikipedia cites the *original* attribution for Vim: "At the time of its first release, the name 'Vim' was an acronym for 'Vi IMitation.'"

and, correspondingly, with editing needs. For instance, its context-specific language editing started with C and has grown to encompass C++, Java, and now C#.

Vim includes many features that facilitate the editing of code in many new languages. As the computing landscape changes, Vim evolves with it.

Today Vim is so ubiquitous, especially among Unix and its variants (e.g., BSD and GNU/Linux), that for many (if not most) users Vim is synonymous with `vi`. Indeed, most if not all distributions of GNU/Linux come with a default installation of Vim as the */usr/bin/vi* binary!

Vim provides features not in `vi` that are considered essential in modern-day text editors, such as ease of use,[2] graphical terminal support, color, and syntax highlighting and formatting, as well as extended customization.

Overview

This section provides an overview of Vim and many of its enhancements, with cross-references to where in the book those enhancements are described.

Author and History

Bram started work on Vim after buying an Amiga computer in the second half of 1988.[3] As a Unix user, he'd been using the `vi`-like editor called `stevie`, one he considered far from perfect. Fortunately, it came with the source code, and he began making the editor more compatible with `vi` and fixing bugs. After a while the program became quite usable. The first version of Vim was created on November 2, 1991, and was published in January 1992; Vim version 1.14 was published on Fred Fish disk 591 (a collection of free software for the Amiga).

Other people began to use the program, liked it, and started helping with its development. A port to Unix was followed by ports to MS-DOS and other systems, and subsequently Vim became one of the most widely available `vi` clones. More features were added gradually: multilevel undo, multiwindowing, and numerous others. Some features were unique to Vim, but many were inspired by other `vi` clones. The goal was and remains to provide the best features to the user.

Today Vim is probably the most full-featured of all the `vi`-style editors. The online help is extensive.

---

2 Ease of use is subjective, but we strongly believe that users who invest the time to learn Vim's extended features will agree with this assessment.

3 This section is adapted from material supplied by Bram Moolenaar, Vim's author. We thank him. You can find more information on Vim's history in Bram's "Vim 25" talk.

One of the more obscure features of Vim is its support for typing from right to left, which is useful for languages such as Hebrew and Farsi and which illustrates Vim's versatility.

Being a rock-stable editor on which professional software developers can rely is another of Vim's design goals. Vim crashes are rare, and when they happen, you can recover your changes.

Development on Vim continues. The group of people helping to add features and port Vim to more platforms is growing, and the quality of the ports to different computer systems is increasing. The Microsoft Windows version has dialogs and a file-selector, which opens up the hard-to-learn `vi` commands to a large group of users.

Why Vim?

Vim so dramatically extends the traditional `vi` functionality that one might more easily ask, "Why *not* Vim?" `vi` introduced the standard from which other clones borrowed, and Vim took the baton and ran with it. Vim dared to radically extend features, sometimes pushing processors to the edge of their ability to perform Vim's work with adequate response time. We don't know whether it was an article of faith by Bram that processor and memory speeds would improve enough to catch up with Vim's demands, but fortunately, modern processors and computers handle even the toughest Vim tasks well.

Compare and Contrast with vi

Vim is more universally available than `vi`. There is at least some version of Vim available on virtually all operating systems, whereas `vi` is available only on Unix or Unix work-alike systems.

`vi` is the original and has changed little over the years. It is the POSIX standard-bearer and fulfills its role well. Vim starts where `vi` leaves off, providing all of `vi`'s functionality and then extending that to add graphical interfaces and features such as complex options and scripting that go far beyond `vi`'s original capabilities.

Vim ships with its own built-in documentation in the form of a directory of specialized text files. A casual inspection of this directory (using the standard Unix word count tool, `wc -l *.txt`) shows 140 files comprising almost 200,000 lines of documentation![4] This is the first hint at the scope of Vim's features. Vim accesses these files via its internal "help" command, another feature not available in `vi`. We

---

4 You can find these files from your Vim session. The help directory is *doc* under the $VIMRUNTIME directory. Type the ex command `:!ls $VIMRUNTIME/doc`.

look more closely at Vim's help system later and offer tips and tricks to maximize your learning experience.

In the current version, Vim's *vi_diff* help file gives an overview of how Vim differs from the original `vi`. The details got to be so numerous that the help files' annotations of what was "not in `vi`" created too much clutter and were removed!

This and the following chapters cover some of the more interesting Vim features. From extensions of the historic `vi` to new functionality, we describe the best and most popular productivity features. We cover topics universally recognized as useful enhancements, such as syntax color highlighting. We also look at some more obscure features that are useful for added productivity. For example, in the section "Autocommands" on page 300, we show a way to customize the Vim status line to show a real-time update of the date and time whenever you move the cursor.

Categories of Features

Vim's features span the range of activities common to virtually any text-editing task. Some features just extend what users wanted the original `vi` to do; others are completely new and not in `vi`. And if you need something that's *not* there, Vim offers built-in scripting for unlimited extensibility and customization. Some categories of Vim features include:

Initialization
   Vim, like `vi`, uses configuration files to define sessions at startup time, but Vim has a greatly expanded repertoire of definable behaviors. You can keep it as simple as setting a few options, as you would in `vi`, or you can write an entire suite of customizations that define your session based on any context you define. For example, you can script your initialization files to precompile code based on which directory you're editing files in, or you can retrieve information from some real-time source and incorporate it into your text at startup. See the section "Startup and Initialization Options" on page 167 in this chapter.

Infinite undo
   Unix `vi` allows you to undo only your last change or to restore the current line to the state it was in before you started making any changes. Vim provides "infinite undo," the ability to keep undoing your changes, all the way back to the state the file was in before you started *any* editing. See the section "Extended Undo" on page 180, later in this chapter, for more information.

Graphical user interface (GUI) features
   Vim extends usability to a more general population by allowing point-and-click editing, like many modern easy-to-use editors. All of the power-user functionality gets the boost of simple GUI accessibility for lighter and simpler editing tasks. See Chapter 9, "Graphical Vim (gvim)", for more information.

Multiple windows

As mentioned already, Vim provides the ability to have multiple windows open simultaneously, on the same file and on different files. This is discussed fully in Chapter 10, "Multiple Windows in Vim".

Programmer assistance

Although Vim doesn't try to provide all programming needs, it offers many features normally found in integrated development environments (IDEs). From quick edit-compile-debug cycles to autocompletion of keywords, Vim has specialized features to let you do more than edit quickly—it helps you program. See Chapter 11, "Vim Enhancements for Programmers", for more information.

Of course, if you want to, you *can* turn Vim into an IDE; see Chapter 15, "Vim as IDE: Some Assembly Required", for more information.

Keyword completion

Vim lets you complete partially typed words with context-sensitive completion rules. For example, Vim can look up words in a dictionary or in a file containing keywords specific to a language. This is discussed in the section "Keyword and Dictionary Word Completion" on page 260.

Syntax extensions

Vim lets you control indentation and syntax-based color coding of your text. And you have many options to define this automatic formatting. If you don't like the color highlighting, you can change it. If you need a certain style of indentation, Vim provides it, or if you have a specialized need, it lets you customize your environment. See the section "Syntax Highlighting" on page 271 for more information.

Scripting and plug-ins

You can write your own Vim extensions or download plug-ins from the internet. You can even contribute to the Vim community by publishing your extensions for others to use. See Chapter 12, "Vim Scripts", for more information.

Postprocessing

In addition to performing initialization functions, Vim lets you define what to do *after* you've edited a file. You can write cleanup routines to delete temporary files accumulated from compiles, or do real-time edits to the file before it's written back to storage. For example, you can check Python code layout for consistent adherence to local formatting rules. You have complete control to customize any postediting activities.

Arbitrary length lines and binary data

Historic versions of `vi` often had limits of around one thousand characters per line; longer lines would be truncated. Vim handles lines of any length.[5]

Vim is Unicode-aware and displays multibyte encoded Unicode characters as a single glyph when possible. Vim is also 8-bit clean, meaning that it can edit files containing any 8-bit character. You can even edit binary and executable files, if necessary. This can be really useful at times. Editing binary files is discussed in the section "Editing Binary Files" on page 318. Also see the section "Vim 8.2 Options" on page 464 for more information about options, specifically `file format` and `filetype`.

There is one tricky detail. Traditional `vi` always writes the file with a final newline appended. When editing a binary file, this might add one character to the file and cause problems. Vim is compatible with `vi` by default and adds that newline. You can set the `binary` option so that this doesn't happen.

Session context

Vim keeps session information in a file, *.viminfo*. Ever wonder "Where was I?" when revisiting and editing a file? The *.viminfo* configuration file fixes that! You can define how much and what kind of information to preserve across sessions. For example, you can define how many "recent documents" or last-edited files to track, how many edits (deletions, changes) to remember per file, how many commands to remember from the command history, and how many registers and lines to keep from previous editing actions ("puts," "deletes," etc.).

Vim also remembers which line you were on for each of your most recently edited files. If you exit your editing session with the cursor on line 25, it repositions you on line 25 the next time you edit that file. See the section "viminfo: Now, Where Was I?" on page 330 for more information.

Transitions

Vim manages state transitions. When you move within a session from buffer to buffer or window to window (usually the same thing), Vim automatically does pre- and postaction housekeeping.

Transparent editing

Vim detects and automatically unbundles archived or compressed files. For example, you can directly edit a compressed file such as *myfile.txt.gz*. You can even edit directories. Vim lets you navigate a directory and select files to edit using familiar `vi`-style navigation commands.

---

5 Well, up to the maximum value of a C `long`—2,147,483,647 on a 32-bit computer, considerably more on a 64-bit system.

Meta-information

Vim offers four handy read-only registers from which the user may extract meta-information for "puts": the current filename (%), the alternate filename (#), the last command-line command executed (:), and the last inserted text (., a period).

The black hole register

This is an obscure but useful extension of editing registers. Normally, text deletions put the deleted text into registers using a rotation scheme (see the section "Making Use of Registers" on page 60), which is useful for cycling through old deletes to get back old and deleted text. Vim provides the "black hole" register as a place to throw deleted text away without affecting the rotation of deleted text in the normal registers. If you're a Unix user, this register is Vim's version of */dev/null*. From the Vim help file *change.txt*:

> *Black hole register* "_
> When writing to this register, nothing happens. This can be used to delete text without affecting the normal registers. When reading from this register, nothing is returned.

Vim also lets you drop back to a `vi`-compatible mode with its `compatible` option (`:set compatible`). Most of the time you'll probably want Vim's extra features, but it's a thoughtful touch to provide for backward compatibility if you need it.

Philosophy

Vim's philosophy aligns closely with `vi`'s. Both provide power and elegance in editing. Both rely on modality (command mode versus input mode). And both keep editing at the keyboard; that is, you can perform all of your editing work quickly and efficiently and never touch a mouse. We like to think of this as "touch editing," which is analogous to "touch typing," reflecting the corresponding increase in speed and efficiency that both bring to their respective tasks.

Vim extends that philosophy by providing features for less experienced users (GUI, visual highlight mode) and giving power options for the power users (scripting, extended regular expressions, configurable syntax, and configurable indenting).

And for the super power users who like to code, Vim comes with source code. Users are free (and even encouraged) to improve on the improvements. Philosophically, Vim strikes a balance for *all* users' needs.

Aids and Easy Modes for New Users

Recognizing that both `vi` and Vim make some learning demands on new users, Vim provides several features that make it easier to use and learn:

Graphical Vim (`gvim`*)*

> When you invoke the `gvim` command, Vim display a rich graphical window, offering full Vim functionality with the addition of the point-and-click features made popular by modern GUI programs. In many environments, `gvim` is a different binary file created by compiling Vim with all of the GUI options turned on. It can also be invoked through `vim -g`.

"Easy" Vim (`evim`*)*

> The `evim` command substitutes some simple behaviors for standard `vi` features, which users who are unfamiliar with `vi` might find to be a more intuitive way to edit files. Expert users probably won't find this mode easy, because they're already used to standard `vi` behavior. It can also be invoked through `vim -y`.

`vimtutor`

> Vim comes with `vimtutor`, a separate command that essentially starts Vim with a special help file. This invocation of Vim gives users another starting point for learning the editor. `vimtutor` takes about 30 minutes to complete.

You can also find any number of interactive Vim tutorials on the internet. One such tutorial is OpenVim. Another is VIM Adventures, which treats learning Vim as an adventure game.

Built-In Help

As mentioned earlier, Vim comes with close to two hundred thousand lines of documentation. Almost all of this documentation is immediately available to you from Vim's built-in help facility. In its simplest form, you invoke the `:help` command. This is interesting because it exposes users to their first example of Vim's multiple window editing.

While this is nice, it presents a bit of a chicken-and-egg conundrum because the built-in help requires a modicum of understanding of `vi` navigation techniques; for it to be really effective, users must know how to jump back and forth in tags. We'll give an overview of help screen navigation here.

The `:help` command brings up something similar to:

```
    *help.txt*      For Vim version 8.2.  Last change: 2020 Aug 15

                    VIM - main help file
                                                                 k
      Move around:  Use the cursor keys, or "h" to go left,    h   l
                    "j" to go down, "k" to go up, "l" to go right.    j
 Close this window: Use ":q<Enter>".
   Get out of Vim:  Use ":qa!<Enter>" (careful, all changes are lost!).

 Jump to a subject: Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
    With the mouse:  ":set mouse=a" to enable the mouse (in xterm or GUI).
```

```
                      Double-click the left mouse button on a tag, e.g. |bars|.
        Jump back:    Type CTRL-O.  Repeat to go further back.

 Get specific help:   It is possible to go directly to whatever you want help
                      on, by giving an argument to the |:help| command.
                      Prepend something to specify the context:  *help-context*

                          WHAT                PREPEND    EXAMPLE      ~
                        Normal mode command              :help x
                        Visual mode command     v_       :help v_u
                        Insert mode command     i_       :help i_<Esc>
                        Command-line command    :        :help :quit
                        Command-line editing    c_       :help c_<Del>
                        Vim command argument    -        :help -r
                        Option                  '        :help 'textwidth'
                        Regular expression      /        :help /[
                      See |help-summary| for more contexts and an explanation.

  Search for help:    Type ":help word", then hit CTRL-D to see matching
                      help entries for "word".
                      Or use ":helpgrep word". |:helpgrep|

  Getting started:    Do the Vim tutor, a 30-minute interactive course for the
                      basic commands, see |vimtutor|.
                      Read the user manual from start to end: |usr_01.txt|

 Vim stands for Vi IMproved.  Most of Vim was made by Bram Moolenaar, but only
 through the help of many others.  See |credits|.
```

Thankfully, Vim accommodates the potential navigation problem for beginners and considerably opens with basic guidelines for navigation, and it even tells you how to exit the help screen. We recommend this as a starting point and urge you to spend time exploring the help.

Once you are familiar with `:help`, you can branch out by using tab completion in Vim's command line. For any command at the command prompt (`:`), pressing TAB results in context-sensitive command-line completion. For example, the following:

```
:e /etc/pas TAB
```

on any Unix system would expand to:

```
:e /etc/passwd
```

The `:e` command implies that the command argument is a file, so command completion looks for files that match the partial filename to complete the input.

But `:help` has its own context, covering the help topics. The partial topic string you type is matched by a substring in any available Vim help topic. We strongly encourage you to learn and use this feature. It saves time and reveals new and interesting features you probably didn't know about.

For example, suppose you want to know how to split a screen. Start with:

```
:help split
```

and press TAB . In the current session, the help command cycles through: `split(); :split; :split_f; splitfind; splitview; g:netrw_browse_split; :diff split; :dsplit; :isplit; :vsplit; +vertsplit; 'splitright'; 'splitbelow';` and many more. To see help for any topic, press the ENTER key when that topic is highlighted. You'll not only see what you're probably looking for (`:split`), but you will also discover things you didn't realize you could do, such as `:vsplit`, the "vertical split" command.

Startup and Initialization Options

Vim uses different mechanisms to set up its environment at startup. It inspects command-line options. It also self-inspects (how was it invoked, and by what name?). There are different compiled binaries to serve different needs (GUI versus text window). Vim also uses a sequence of initialization files in which uncountable combinations of behaviors can be defined and modified. There are too many options to cover completely; we will touch on some of the interesting ones. In the next sections, we discuss Vim's starting sequence in the following order:

- Command-line options
- Behaviors associated to command name
- Configuration files (system-wide and per-user)
- Environment variables

This section introduces you to *some* of the ways to start Vim. For a more detailed discussion of many more options, use the help command:

```
:help startup
```

Command-Line Options

Vim's command-line options provide flexibility and power. Some options invoke extra features, whereas others override and suppress default behavior. We discuss the command-line syntax as it would be used in a typical Unix environment. Single-letter options begin with - (one hyphen), as in `-b`, which allows editing of binary files. Word-length options begin with `--` (two hyphens), as in `--noplugin`, which overrides the default behavior of loading plug-ins. A command-line argument of two hyphens by themselves tells Vim that the rest of the command line contains no options. This is standard Unix behavior.

Following the command-line options, you can optionally list one or more filenames to be edited. Actually, there is an interesting case in which a filename can be a single hyphen, telling Vim that input comes from the standard input, *stdin*, but this is an advanced usage.

The following is a partial list of Vim command-line options not available in vi (all vi options are available in Vim):

`-b`

Edit in binary mode. This is self-explanatory and very cool. Editing binary files is an acquired taste, but this is a powerful way to edit files not touchable by most other tools. If interested, you should read Vim's help section on editing binary files.

`-c` *command*

Execute *command* as an ex command. vi has this same option, but Vim allows up to 10 `-c` instances in one command. Each command instance must use its own `-c` flag.

`-C`

Run Vim in compatible (vi) mode. For obvious reasons, this option would never be in vi.

`--cmd` *command*

Execute *command* before *vimrc* files. This is the long form of the `-c` option.

`-d`

Start in diff mode. Vim performs a diff on two, three, or four files and sets options making inspection of file differences simple (`scrollbind`, `foldcolumn`, etc.).

Vim uses the native file difference program, which is `diff` on Unix systems. The Windows version offers a downloadable executable with which Vim can perform the diff.

`-E`

Start in improved ex mode. For example, improved ex mode would use extended regular expressions.

`-F` *or* `-A`

Farsi or Arabic modes, respectively. These require key and character maps to be useful and draw the screen from right to left.

`-g`

Start `gvim` (GUI).

`-M`

Turn off the write option. Buffers will not be modifiable. While you can't modify the buffer, Vim ensures no changes by disabling both the `:w` and `:w!` ex commands.

-o*[n]*

> Open all files in separate windows. Optionally an integer can specify the number of windows to open. Files named on the command line fill that number of windows only (the rest are in Vim buffers). If the specified number of windows exceeds the listed files, Vim opens empty windows to satisfy the requested count of windows.

-O*[n]*

> Like `-o`, but opens vertically split windows.

-y

> Run Vim in "easy" mode. This sets options to a more intuitive behavior for beginners. While "easy" may help the uninitiated, seasoned users will find this mode confusing and irritating.

-Z

> Run in restricted mode. This basically turns off all external interfaces and prevents access to the system features. For example, users can't use `!G!sort` to sort from the current line in the buffer to the end of the file; the filter `sort` will not be available.

The following is a series of related options to use a remote instance of a server Vim. The `--remote` options tell a remote Vim (which may or may not be executing on the same machine) to edit a file or evaluate an expression in that remote server. The `--server` options tell Vim which server to send to or indicate that Vim can declare itself as a server. `--serverlist` simply lists available servers:

> `--remote` *file*
> `--remote-expr` *expr*
> `--remote-send` *keys*
> `--remote-silent` *file*
> `--remote-tab`
> `--remote-tab-silent`
> `--remote-tab-wait`
> `--remote-tab-wait-silent`
> `--remote-wait` *file* ...
> `--remote-wait-silent` *file* ...
> `--serverlist`
> `--servername` *name*

For a more complete discussion of all command-line options, including the complete vi set, refer to the section "Command-Line Syntax" on page 415. For more information on the `--remote` options, issue the Vim command `:help remote`.

_

Behaviors Associated to Command Name

Vim comes in two main flavors: graphical (using the X Window System under Unix variants and native GUIs in other operating systems) and textual, each of which can start up with subsets of characteristics. Unix users simply use one of the commands in the following list to get the desired behavior:

`vim`

Start the text-based Vim.

`gvim`

Start Vim in graphical mode. In many environments, `gvim` is a different binary file of Vim with all of the GUI options turned on during compilation. This has the same effect as starting Vim with `vim -g`.

`view, gview`

Start Vim or `gvim` in read-only mode. This has the same effect as starting Vim with `vim -R`.

`rvim`

Start Vim in restrictive mode. All external access to shell commands is disabled, as well as the ability to suspend the editing session with the `^Z` command.

`rgvim`

This has the same effect as starting Vim with `rvim` but for the graphical version.

`rview`

Analogous to `view`, but start in restricted mode. In restricted mode, users do not have access to filters, outside environments, or OS features. This has the same effect as starting Vim with `vim -Z` (the `-R` option invokes just the read-only effect described previously).

`rgview`

This has the same effect as starting Vim with `rview` but for the graphical version.

`evim, eview`

Use "easy" mode for editing or read-only viewing. Vim sets options and features so it behaves in a more intuitive way for those who are not familiar with the Vim paradigm. This has the same effect as starting Vim with `vim -y`. Expert users probably won't find this mode easy because they're already used to standard `vi` behavior.

Note there is no analogous *gXXX* version of these commands, because `gvim` is ostensibly thought to be already easy, or at least intuitive to learn, with predictable point-and-click behavior.

```
vimdiff, gvimdiff
```
Start in "diff" mode and perform a diff on the input files. This is covered in depth later in the section "What's the Difference?" on page 328.

```
ex, gex
```
Use line-editing `ex` mode. Useful in scripts. This has the same effect as starting Vim with `vim -e`.

MS-Windows users can access a similar choice of Vim versions in the program list (Start menu).

System and User Configuration Files

Vim looks for initialization cues in a special sequence. It executes the first set of instructions it finds (either in the form of an environment variable or in a file) and begins editing. Thus, the first element of the following list that is encountered is the only element of the list that is executed. The sequence follows:

1. `VIMINIT`: This is an environment variable. If it is nonempty, Vim executes its content as an `ex` command.

2. User *vimrc* files: The *vimrc* (Vim resource) initialization file is a cross-platform concept, but because of subtle operating system and platform differences, Vim looks for it in different places in the following order:

$HOME/.vimrc	Unix, OS/2,[a] and Mac OS X
$HOME/_vimrc	MS-Windows and MS-DOS
$VIM/_vimrc	MS-Windows and MS-DOS
s:.vimrc	Amiga
home:.vimrc	Amiga
$VIM/.vimrc	OS/2 and Amiga

    [a] We don't know how many people still have OS/2 or Amiga systems. But if you do, it's nice to know that Vim supports you!

3. Local *.exrc* and *vimrc* files: If the Vim `exrc` option is set, Vim looks for the three additional configuration files: *.vimrc*; *.gvimrc*; and *.exrc*. On non-POSIX systems, the filename may start with something different than a period.

The *.vimrc* file is a good place to configure Vim's editing characteristics. Virtually any Vim option can be set or unset in this file, and it is particularly suited to setting up global variables and defining functions, abbreviations, key mappings, and so forth. Here are a few things to know about the *.vimrc* file:

- Comments begin with a double quote ("), and the double quote can be anywhere in the line. All text after and including the double quote is ignored.

- `ex` commands can be specified with or without a colon. For example, `set autoindent` is identical to `:set autoindent`.

- The file is much more manageable if you break large sets of option definitions into separate lines. For example:

```
set terse sw=1 ai ic wm=15 sm nows ruler wc=<Tab> more
```

is equivalent to:

```
set terse       " short error and info messages
set shiftwidth=1
set autoindent
set ignorecase
set wrapmargin=15
set nowrapscan  " don't scan past end or top of file in searches
set ruler
set wildchar=<TAB>
set more
```

Notice how much more readable the second set of commands is. The second method is also much easier to maintain through deletions, insertions, and temporarily commenting out lines when debugging settings in the configuration file. For example, should you want to temporarily disable line numbering in the startup configuration, you simply insert a double quote (") at the beginning of the `set number` line in your configuration file.

Environment Variables

Many environment variables affect Vim's startup behavior and even some editing session behavior. These are mostly transparent and are handled with defaults if not configured.

How to set environment variables

The command environment you have when you log in (called the shell in Unix) sets variables to reflect or control its behavior. Environment variables are especially powerful because they affect programs invoked within the command environment. The following instructions are not specific to Vim; they can be used to set any environment variables you want set in the command environment:

MS-Windows
    To set an environment variable:

    1. Bring up the control panel.

    2. Double-click `System`.

3. Click the `Advanced` tab.

4. Click the `Environment Variables` button.

The result is a window divided into two environment variable areas, `User` and `System`. Novices shouldn't modify the `System` environment variables. In the `User` area, you can set environment variables related to Vim and make them persist across login sessions.

Unix/Linux Bash and other Bourne shells
Edit the appropriate shell configuration file (such as *.bashrc* for Bash users) and insert lines resembling:

```
VARABC=somevalue VARXYZ=someothervalue MYVIMRC=/path/to/my/vimrc/file
export VARABC VARXYZ MYVIMRC
```

The order of these lines is irrelevant. The `export` statement just makes variables visible to programs that run in the shell, and thus turns them into environment variables. The value of exported variables can be set before or after exporting them.

Unix/Linux C shells
Edit the appropriate shell configuration file (such as *.cshrc*) and insert lines resembling the following:

```
setenv VARABC somevalue
setenv VARXYZ someothervalue
setenv MYVIMRC /path/to/my/vimrc/file
```

Environment variables relevant to Vim

The following list shows most of Vim's environment variables and their effects.

The Vim `-u` command-line option overrides Vim's environment variables and goes directly to the specified initialization file. The `-u` does *not* override non-Vim environment variables:

EXINIT
Same as `VIMINIT`; used if `VIMINIT` isn't defined.

MYVIMRC
Overrides Vim's search for initialization files. If `MYVIMRC` has a value when starting, Vim assumes the value is the name of an initialization file and, if the file exists, takes initial settings from it. No other file is consulted (see the search sequence in the previous section).

SHELL

Specifies which shell or external command interpreter Vim uses for shell commands (`!!`, `:!`, etc.). In an MS-Windows command window, if `SHELL` is not set, the `COMSPEC` environment variable is used instead.

TERM

Sets Vim's internal `term` option. This is somewhat unnecessary, because the editor sets its terminal itself as it deems appropriate. In other words, Vim probably knows what the terminal is better than a predefined variable.

VIM

Contains the path of a system directory where standard Vim installation information is found (for information only and not used by Vim).

VIMINIT

Specifies `ex` commands to execute when Vim starts. Define multiple commands by separating them with vertical bars (`|`).

> If more than one version of Vim exists on a machine, `VIM` will likely reflect different values depending on which version you start. For example, on one author's machine, the Cygwin version sets the `VIM` environment variable to */usr/share/vim*, whereas the `vim.org` package sets it to *C:\Program Files\Vim*.
>
> This is important to know if you are making changes to Vim files, as changes may not take effect if you edit the wrong files!

VIMRUNTIME

Points to Vim support files, such as online documentation, syntax definitions, and plug-in directories. Vim typically figures this out on its own. If you set the variable—for example, in the *.bashrc* file—it can cause errors if a newer version of Vim is installed because your personal `VIMRUNTIME` variable may point to an old, nonexistent, or invalid location.

New Motion Commands

Vim provides all `vi` movement or motion commands, most of which are listed in Chapter 3, "Moving Around in a Hurry", and adds several others, summarized in Table 8-1.

Table 8-1. Motion commands in Vim

Command	Description
n CTRL-END	Go to the end of the file, i.e., the last character of the last line of the file. If *n* was specified, go to the last character of the line *n*.
HOME	Go to the first nonblank character of the first line of the file. This differs from CTRL-END in that HOME does not move the cursor to whitespace.
count %	Go to the line *count* percent into the file, putting the cursor on the first nonblank line. It's important to note that Vim bases its calculation on the number of lines in the file, not the total character count. This may not seem important, but consider an example of a file containing 200 lines, of which the first 195 contain 5 characters (for example, prices such as `$4.98`), and the last 4 lines contain 1,000 characters. In Unix, accounting for the newline character, the file would contain approximately: (195 * (5 + 1)) *(The number of characters in the first five-character lines)* + 2 + (4 * (1000 + 1)) *(The number of characters in the thousand-character lines)* or 5,200 characters. A by-character 50% count would place the cursor on line 96, whereas Vim's 50% motion command places the cursor on line 100.
`:go` *n* `:`*n* `go`	Go to the *n*th byte in the buffer. All characters, including end-of-line characters, are counted.

The notation `<C-xxx>` is Vim's way of describing key combinations in a system-independent manner. In this case, the leading `C-` means to hold the CTRL key while pressing the other key for *xxx*. For example, `<C-End>` means press CTRL and END.

Visual Mode Motion

Vim lets you define selections visually and perform editing commands on the visual selection. This is similar to what you may see in graphical editors in which you highlight areas by clicking and dragging the mouse. What Vim offers with its visual mode is the convenience of seeing the selection on which work is done *and* all of the powerful Vim commands with which to do work on the visually selected text. This lets you do much more sophisticated work on highlighted text than the traditional cut and paste actions in less advanced editors.

You can select a visual area in Vim in the same manner as other editors, by clicking and dragging the mouse. But Vim also lets you use its powerful motion commands and some special visual mode commands to make the selection.

For example, you can type `v` in command mode to start visual mode. Once you are in visual mode, any motion commands move the cursor *and* highlight text as the cursor moves to a new position. So, the "next word" command (`w`) in visual mode moves the cursor to the next word and highlights the selected text. Additional movements extend the selected region appropriately.

In visual mode, Vim uses some specialized commands with which you conveniently extend the selected text by selecting the text object around the cursor. For example, the cursor can be within a "word," and at the same time be within a "sentence," and

also be within a "paragraph." Vim lets you add to the visual selection with commands that extend the highlighted region to a text object. To visually select a word, you use `aw` (when in visual mode).

You can highlight visual areas of the buffer in several ways. In text-based mode, simply type `v` to toggle visual mode on and off. When on, visual mode selects and highlights the buffer as the cursor moves. In `gvim`, just click and drag the mouse across the desired region. This sets Vim's visual flag. Table 8-2 shows some of Vim's visual mode motion commands.

Table 8-2. Visual mode motion commands in Vim

Command	Description
n aw, *n* aW	Select *n* words. Intervening whitespace is included. This is slightly different from `iw` (see next entry). Lowercase `w` looks for punctuation-delimited words, whereas uppercase `W` looks for whitespace-delimited words.
n iw, *n* iW	Select *n* words. Add words but not whitespace. Lowercase `w` looks for punctuation-delimited words, whereas uppercase `W` looks for whitespace-delimited words.
as, is	Add sentence, or inner sentence.
ap, ip	Add paragraph, or inner paragraph.

For a more detailed discussion of text objects and how they are used in visual mode, use the help command:

```
:help text-objects
```

We recommend that you play around with visual mode and get used to it. In particular, it's a great way to choose the text to which you wish to apply a substitute command, or when you want send text through a filter.

Extended Regular Expressions

The metacharacters available in `vi`'s search and substitution regular expressions are described back in Chapter 6 in the section "Metacharacters Used in Search Patterns" on page 90.

Vim provides extended regular expressions, which are always available. Some of these additional metasequences give power equivalent to that provided by `egrep` (or `grep -E` on fully POSIX-compliant systems). Some of the descriptive text in the following list is borrowed from the Vim documentation:

`\|`

Indicates alternation. For example, `a\|b` matches either *a* or *b*. However, this construct is not limited to single characters: `house\|home` matches either of the strings *house* or *home*.

\\&

> Indicates a "concat." A concat matches the part after the last `\&`, but only if all the previous parts matched. The Vim help gives these examples: `foobeep\&...` matches *foo* in *foobeep*, and `.*Peter\&.*Bob` matches in a line containing both *Peter* and *Bob*.

\\+

> Match one or more of the preceding regular expressions. This is either a single character or a group of characters enclosed in parentheses. Note the difference between `\+` and `*`. The `*` is allowed to match nothing, but with `\+` there must be at least one match. For example, `ho(use\|me)*` matches *ho* as well as *home* and *house*, but `ho(use\|me)\+` does not match *ho*.

\\=

> Match zero or one of the preceding regular expression. This is the same as `egrep`'s ? operator.

\\?

> Match zero or one of the preceding regular expression. This is the same as `egrep`'s ? operator and is more natural for people familiar with `egrep` and `awk`.

\\{...}

> Defines an *interval expression*. Interval expressions describe counted numbers of repetitions. Vim requires only the left brace to be preceded by a backslash, not the right brace. In the following descriptions, *n* and *m* represent integer constants:

> **\\{*n,m*}**

>> Match *n* to *m* of the preceding regular expression, as much as possible. The bounding is important, since it controls how much text would be replaced during a substitute command.[6] *n* and *m* are nonnegative numbers (this includes zero).

> **\\{*n*}**

>> Match exactly *n* repetitions of the previous regular expression. For example, `(home\|house){2}` matches *homehome*, *homehouse*, *househome*, and *house-house*, but nothing else.

---

6 The `*`, `\+`, and `\=` operators can be reduced to `\{0,}`, `\{1,}`, and `\{0,1}`, respectively, but the former are much more convenient to use. Also, interval expressions were developed later in the history of Unix regular expressions.

\\{*n*,}
> Match at least *n* of the preceding regular expression, as much as possible. Think of it as "as least *n*" repetitions.

\\{,*m*}
> Match zero to *m* of the preceding regular expression, as much as possible.

\\{}
> Matches zero or more of the preceding regular expression, as much as possible (same as *).

\\{-*n*,*m* }
> Matches *n* to *m* of the preceding regular expression, as few as possible.

\\{-*n*}
> Matches *n* of the preceding regular expression.

\\{-*n*,}
> Matches at least *n* of the preceding regular expression, as few as possible.

\\{-,*m*}
> Matches zero to *m* of the preceding regular expression, as few as possible.

~

Matches the last given substitute (i.e., replacement) string.

\\(…\\)

Provides grouping for *, \\+, \\?, and \\=, as well as making matched subtexts available in the replacement part of a substitute command (\\1, \\2, etc.).

\\1

Matches the same string that was matched by the first subexpression in \\(and \\). For example, \\([a-z]\\).\\1 matches *ata*, *ehe*, *tot*, etc. \\2, \\3, and so on may be used to represent the second, third, and so on subexpressions.

The `isident`, `iskeyword`, `isfname`, and `isprint` options define the characters that appear in identifiers, keywords, and filenames, and that are printable. Use of these options makes regular expression matching very flexible.

Vim provides a number of additional special sequences that act as shorthands for several nonprinting characters, as well as for commonly used bracket expressions. Vim calls these *character classes* after the earlier, original usage in Unix documentation. They are presented in Table 8-3.

Table 8-3. Vim regular expression characters and character classes

Sequence	Meaning
\a	Alphabetic character: same as [A-Za-z].
\A	Nonalphabetic character: same as [^A-Za-z].
\b	Backspace.
\d	Digit: same as [0-9].
\D	Nondigit: same as [^0-9].
\e	Escape.
\f	Matches any filename character, as defined by the isfname option.
\F	Like \f, but excluding digits.
\h	Head of word character: same as [A-Za-z_].
\H	Non-head-of-word character: same as [^A-Za-z_].
\i	Matches any identifier character, as defined by the isident option.
\I	Like \i, but excluding digits.
\k	Matches any keyword character, as defined by the iskeyword option.
\K	Like \k, but excluding digits.
\l	Lowercase character: same as [a-z].
\L	Nonlowercase character: same as [^a-z].
\n	Matches a newline. Can be used to match multiline patterns.
\o	Octal digit: same as [0-7].
\O	Non-octal digit: same as [^0-7].
\p	Matches any printable character, as defined by the isprint option.
\P	Like \p, but excluding digits.
\r	Carriage return.
\s	Matches a whitespace character (exactly a space or a tab).
\S	Matches anything that isn't a space or a tab.
\t	Matches a tab.
\u	Uppercase character: same as [A-Z].
\U	Nonuppercase character: same as [^A-Z].
\w	Word character: same as [0-9A-Za-z_].
\W	Nonword character: same as [^0-9A-Za-z_].
\x	Hexadecimal digit: same as [0-9A-Fa-f].
\X	Nonhexadecimal digit: same as [^0-9A-Fa-f].
_*x*	Where *x* is any of the previous characters above: match the same character class but with newline included.

Finally, Vim provides quite a number of additional, very esoteric ways to match regular expressions. See `:help regexp` for the full details if you are interested. However, we think that the list and table in this section provide quite enough features to keep you busy for now.

Extended Undo

Beyond the convenience of undoing an arbitrary number of edits, Vim offers an interesting twist called *branching* undos.

To use this feature, first decide how much control you want over undoing edits. Use the `undolevels` option to define the number of undoable changes you can make in an editing session. The default is one thousand, which is probably more than enough for most users. If you want `vi` compatibility, set `undolevels` to zero:

```
:set undolevels=0
```

In `vi`, the undo command (`u`) is basically a toggle between the file's current state and its most recent change. The first *undo* reverts to the state before the last change. The next *undo* redoes the undone change. Vim behaves quite differently, and therefore the commands are implemented differently.

Instead of toggling the most recent change, repeated invocations of Vim's undo rolls back the state of the file through the most recent changes, in order, for as many changes as defined by the `undolevels` option. Because the undo command, `u`, only moves backward, we need a command to roll forward and "redo" changes. Vim does this with the redo command, `:redo`, or the CTRL-R key. The CTRL-R key accepts a numeric prefix to redo several changes at once.

When rolling forward and backward through changes with the redo and undo commands, Vim maintains a map of the file's state and knows when the last possible undo has been performed. When all possible undos are undone, Vim resets the file's *modified* status, which allows quitting without the ! suffix. Although this is a modest benefit for general user interaction, it is more useful for behind-the-scenes scripting where the modified state of the file is important.

For most users, simply undoing and redoing changes is sufficient. But consider a more complex scenario. What if you make seven changes to a file, and undo three? So far, so good, nothing unusual to consider. But now suppose that after undoing three out of seven changes, you then make a change different from the next forward change in Vim's collection of changes; Vim defines that point in the change history as a *branch* from which different paths of changes occur. With that path you can now move back and forth chronologically, with the added twist that at a branch point you can move forward along any of the different paths of recorded changes.

For more complete descriptions of how to navigate changes as a tree, use Vim's help command:

```
:help usr_32.txt
```

For deeper analyses of Vim change-trees, check out some of the undo plug-ins at *https://vimawesome.com/?q=undo*.

Incremental Searching

When *incremental searching* is used, the editor moves the cursor through the file, matching text *as you type* the search pattern. When you finally type ENTER, the search is finished.[7] Vim enables incremental searching with the incsearch option. When enabled, Vim highlights the text that matches what you've typed so far, changing what it highlights as you enter more characters into the search pattern.

If you've never seen it before, it is rather disconcerting at first. However, after a while you get used to it, and eventually you come to wonder how you ever did without it. We strongly recommend adding set incsearch to your *.vimrc* file.

Left-Right Scrolling

By default, vi and Vim wrap long lines around the screen. Thus, a single logical line of the file may occupy multiple physical lines on your screen.

There are times when it might be preferable for a long line to simply disappear off the righthand edge of the screen instead of wrapping. Moving onto that line and then moving to the right would "scroll" the screen sideways.

In Vim, a numeric option (sidescroll, default value zero) controls how much to scroll the screen, and a Boolean option (wrap, default true) controls whether lines wrap or disappear off the edge of the screen. Thus, to get side scrolling, you'd use :set nowrap and give sidescroll a reasonable value such 8 or 16.

Summary

For many years vi was the standard text-editing tool on Unix. vi was almost revolutionary in its time, with its dual mode orientation and touch-edit philosophy. Vim continues where vi left off, and it is the next evolutionary step for powerful editing and text management:

---

7 Emacs has always had incremental searching.

- Vim extends `vi`, building on the excellent standard set by the older editor. Although other editors have also built upon the original, Vim has emerged as the most popular and widely used `vi` clone.

- Vim offers far (far!) more than `vi`, so much more as to become the new standard. Vim is the de facto standard in that most Unix-like operating systems link the `vi` command to Vim.

- Vim is for beginners *and* for power users. For beginners, it offers various learning tools and "easy" modes, whereas for experts it offers powerful extensions to `vi`, along with a platform on which power users can enhance and tune Vim to their exact needs.

- Vim runs everywhere. As discussed earlier, in environments in which Vim wasn't available, others stepped in and ported it to most useful OS platforms. Vim may not literally be everywhere, but it's close!

- Vim is free. Furthermore, Vim is charityware. The work Bram Moolenaar has done creating, improving, maintaining, and sustaining Vim is one of the truly remarkable feats in the free software market. If you like *his* work, Bram invites you to learn about his favorite cause, helping children in Uganda. More information is available at the website ICCF Holland, or you can simply use Vim's built-in help command, topic "uganda" (`:help uganda`).

Graphical Vim (gvim)

As a `vi` derivative, Vim began as a project to extend `vi` by adding features not available in `vi`. As an independent effort Vim added and improved on the excellent `vi`, and Vim could do this quickly, based on user feedback, without the onus of POSIX requirements.

Already at the time of this book's seventh edition, Vim offered mature and comprehensive graphical user interface (GUI) features discussed in this chapter. In the years since then, Vim continued enhancing the GUI, and today it is better than ever.

A longtime complaint about `vi` and its clones was their lack of a GUI. Especially for those caught up in the Emacs versus `vi` religious wars, `vi`'s lack of a GUI was the ultimate trump card to argue that `vi` was a nonstarter when discussing editors. That is a complaint long since answered.

The `vi` clones and "work-alikes" created their own GUI versions. Graphical Vim is called `gvim`. Like the other `vi` clones, `gvim` offers robust and extensible GUI functions and features. We'll cover the most useful ones in this chapter.

Some of `gvim`'s graphical functionality wraps commonly used Vim features, whereas others introduce the point-and-click convenience most computer users now expect. Although some veteran Vim users may cringe at the thought of grafting a GUI onto their workhorse editor, `gvim` is thoughtfully conceived and implemented. `gvim` offers functionality and features spanning the range of its users' abilities, softening Vim's steep learning curve for beginners and transparently bringing expert users extra editing power. This strikes a nice compromise.

`gvim` for MS-Windows comes with a menu entry labeled "easy `gvim`." This is indeed valuable to people who have never used Vim, but ironically, it is anything *but* easy for expert users.

In this chapter we first discuss the general `gvim` GUI concepts and features, with a brief introductory section about mouse interaction. Additionally, we refine the discussion around differences and things you should know for different `gvim` environments. Specifically, we focus on MS-Windows and the X Window System, the two main graphical platforms.[1] We also provide a brief list of GUI options with synopses.

General Introduction to gvim

`gvim` brings all the functionality, power, and features of Vim while adding the convenience and intuitive nature of a GUI environment. From traditional menus to visual highlighting, `gvim` provides the GUI experience today's users expect. For veteran, console-based, text-environment `vi` users, `gvim` still gives the familiar core power and doesn't dumb down the paradigm that garnered `vi` its reputation as a power editor.

Starting gvim

You start a graphical session with the `gvim` command or with `vim -g`. In MS-Windows, the self-installing executable adds an "edit with Vim" context-menu entry. This gives quick and easy access to `gvim` by integrating it into the Windows environment.

The configuration files and options recognized by `gvim` are slightly different from those used by Vim. `gvim` reads and executes two startup files: *.vimrc*, followed by *.gvimrc*. Although you can put `gvim`-specific options and definitions in *.vimrc*, it's better to define them in *.gvimrc*. This provides a nice separation of regular Vim and `gvim` customization. It also assures proper behavior on startup. For example, `:set columns=100` isn't valid in regular Vim and generates an error when Vim is started.[2]

---

1 Especially now that MS-Windows offers Windows Subsystem for Linux (WSL), a complete GNU/Linux subsystem with the option for users to install their favorite GNU/Linux distribution. We describe one way to start the native Linux `gvim` within WSL and display the session natively in Windows.

2 We've discovered that Vim doesn't always generate an error and in fact starts with no error messages. The caveat stands because even when Vim doesn't generate an error, Vim tries to reconfigure the definition of your console/screen/terminal accordingly. Some terminals or consoles adapt correctly, but it's more likely that you will end up with a semicorrect behavior of your terminal. The effects can interfere with the correct behavior of other applications that depend on and use terminal definitions.

If a system *gvimrc* file exists (usually in `$VIM/gvimrc`), it is executed. Administrators can use this system-wide configuration file to set common options for their users. This provides a baseline configuration so that users will share a common editing experience.

More experienced Vim users can add their own favorite custom settings and features. After `gvim` reads the optional system configuration, it looks in four places for additional configuration information, in the following order, and stops searching after finding any one of these:

- An `exrc` command stored in the `$GVIMINIT` environment variable.
- A user's *gvimrc* file, usually stored in *$HOME/.gvimrc*. If it is found, it is sourced.
- In a Windows environment, if `$HOME` is not set, `gvim` looks in `$VIM/_gvimrc`. This is the normal situation for Windows users, but it's an important distinction for users who have Unix work-alikes installed and are likely to have the `$HOME` variable set. One example would be the popular Cygwin suite of Unix tools.
- If *_gvimrc* isn't found, `gvim` finally looks for *.gvimrc*.

If `gvim` finds a nonempty file to execute, that file's name is stored in the `$MYGVIMRC` variable and further initialization stops.

There is one more option for customization. If, in the cascading sequence of initialization just described, the option `exrc` is set:

```
:set exrc
```

`gvim` additionally looks in the current directory for *.gvimrc*, *.exrc*, or *.vimrc* and sources that file if it isn't one of the previously listed files (i.e., if it hasn't already been discovered as an initialization file and already executed).

> In a Unix environment, there are security issues around local directories containing configuration files (both *.gvimrc* and *.vimrc*). `gvim` defaults to enforcing some restrictions on what can be executed from these files by setting the `secure` option if the file is not owned by the user. This helps prevent malicious code from being executed. If you want to be sure, set the `secure` option explicitly in your *.vimrc* or *.gvimrc* file. See the section "Vim 8.2 Options" on page 464 for more information on the `secure` option.

Using the Mouse

The mouse in `gvim` does something useful in every editing mode. Let's look at the standard Vim editing modes and how `gvim` treats the mouse in each:

`ex` *command mode*
> You enter this mode when you open the command buffer at the bottom of the window by typing a colon (`:`). If the window is in command mode, you can use the mouse to reposition the cursor anywhere in the command line. This is enabled by default or when you include the `c` flag in the `mouse` option.

Insert mode
> This is the mode for entering text. If you click in a buffer that's in insert mode, the mouse repositions the cursor and lets you immediately start entering text at that position. This mode is enabled by default or when you include the `i` flag in the `mouse` option.
>
> The mouse's behavior in insert mode provides easy and intuitive point-and-click positioning. In particular, it bypasses the need to exit insert mode, navigate with the mouse, motion commands, or other methods, and then reenter insert mode.
>
> Superficially, this seems like a great idea, but in practice it will appeal to only a subset of users. It may be more annoying than helpful to experienced Vim users.
>
> Consider what happens when you are in insert mode and leave `gvim` for some other application. When you click back into the `gvim` window, the point you click is now the insertion point for text, and probably not the one you want. In a single-window `gvim` session, you could land in a different spot from where you were originally working; in a multiple-window `gvim` screen, you could end up with the mouse in a completely different window. You might end up entering text into the wrong file!

`vi` *command mode*
> This includes any time you're not in insert mode or on the command line. Clicking the mouse in the screen simply leaves the cursor on the character where you clicked. This mode is enabled by default or when you include the `n` flag in the `mouse` option.
>
> `vi` command mode provides a straightforward and easy method to position the cursor, but it offers only clunky support for moving beyond the top or bottom of the visible window. Click and hold the mouse and slide to the top or bottom of a window; `gvim` will scroll up and down correspondingly. If scrolling stops, move the mouse back and forth sideways to make it resume. It's not clear why `vi` command mode acts this way.

Another drawback to `vi` command mode is that users, especially beginners, can come to rely on point and click as the positioning method of choice. This can hold back their motivation to learn Vim's navigation commands, and hence its power-editing methods. Finally, it creates the same potential confusion as insert mode.

Additionally, `gvim` offers *visual* mode, also known as *select* mode. This mode is enabled by default, or when you include the v flag in the `mouse` option. Visual is the most versatile mode, because it lets you select text by dragging the mouse, which highlights the selection. It can be used in combination with command, insert, and `vi` command modes.

Any combination of flags can be specified in the `mouse` option. The syntax to use is illustrated by the following commands:

`:set mouse=""`
> Disable all mouse behavior.

`:set mouse=a`
> Enable all mouse behavior (the default).

`:set mouse+=v`
> Enable visual mode (v). This example uses the += syntax to add a flag to the current `mouse` setting.

`:set mouse-=c`
> Disable mouse behavior in `vi` command mode (c). This example uses the -= syntax to remove a flag from the current `mouse` setting.

Beginners may prefer more "on" settings, whereas experts may turn the mouse off completely (as one of us does).

If you use the mouse, we recommend choosing a familiar behavior through `gvim`'s `:behave` command, which accepts either `mswin` or `xterm` as an argument. As suggested by the names of the arguments, `mswin` sets options to closely mimic Windows behavior, whereas `xterm` mimics a window on the X Window System.

Vim has a number of other mouse options, including `mousefocus`, `mousehide`, `mouse model`, and `selectmode`. For more information, refer to the Vim built-in documentation for these options.

If you have a mouse with a scroll wheel, `gvim` handles it well by default, scrolling the screen or window up and down predictably, regardless of how you set the `mouse` option.

Useful Menus

One nice touch `gvim` brings to the GUI environment is menu actions that simplify some of Vim's more esoteric commands. There are two worth mentioning.

gvim's Window menu

`gvim`'s *Window* menu contains many of the most useful and common Vim window management commands: commands that split a single GUI window into multiple display areas. You may find it worth "tearing off" this menu, as shown in Figure 9-1, so that you can conveniently open and bounce around among windows. The result is shown in Figure 9-2. (We discuss tear-off menus shortly.)

Figure 9-1. `gvim`'s Window menu

Notice how the menu in Figure 9-2 is moved and floats over a completely unrelated application. This is a nice way to have an often-used menu conveniently available but out of the way of the editing. Both of these are handy for common select, cut, copy, delete, and paste operations. Users of other GUI editors employ this kind of feature all the time, but this is useful even for longtime Vim users. It is especially useful in that it interacts with the Windows clipboard in a predictable way.

Figure 9-2. gvim's Window menu, torn off and floating

gvim's right-click pop-up menu

gvim pops up the menu shown in Figure 9-3 when you right-click within a buffer you're editing.

Figure 9-3. gvim general editing menu

If any text is selected (highlighted), another menu pops up when you right-click, as shown in Figure 9-4.

Figure 9-4. `gvim` editing menu when text is selected

Customizing Scrollbars, Menus, and Toolbars

`gvim` provides the usual GUI widgets, such as scrollbars, menus, and toolbars. Like most modern GUI applications, these widgets are customizable.

The `gvim` window, by default, shows several menus and a toolbar at the top, as illustrated in Figure 9-5.

Figure 9-5. Top of the `gvim` window (Linux version)

Scrollbars

Scrollbars, which let you quickly navigate up and down or right and left through a file, are optional in `gvim`. You can display or hide them with the `guioptions` option, described at the end of this chapter in the section "GUI Options and Command Synopsis" on page 211.

Because Vim's standard behavior is to show all the text in a file (wrapping lines in the window if necessary), it's interesting to note that the horizontal scrollbar serves no purpose in typically configured `gvim` sessions.

Turn the left and right scrollbars on and off by including or excluding `r` or `l` in the `guioptions` option. `l` makes sure the screen always has a left scrollbar, whereas `r` makes it always have a right scrollbar. The uppercase variants `L` and `R` tell `gvim` to show left or right scrollbars only when there is a vertically split window.

The horizontal scrollbar is controlled by including or excluding `b` in the `guioptions` option.

And yes, you *can* scroll the right and left scrollbars at the same time! More precisely, scrolling either one causes the other to move in the corresponding direction. It can be pretty convenient to have scrollbars configured on both sides. Depending on where your mouse is positioned, you simply click and drag the nearest scrollbar.

> Many options, including `guioptions`, control multiple behaviors, and thus can include many flags by default. New flags could even be added in future versions of `gvim`. Hence, it is important to use the `+=` and `-=` syntax in the `:set guioptions` command to avoid deleting desirable behaviors. For example, `:set guioptions+=l` adds the "scrollbar always on left" option to `gvim`, leaving the other components in the `guioptions` string intact.

Menus

`gvim` has a fully customizable menu feature. In this section we describe the default menu characteristics, which appeared earlier in Figure 9-5, and show how you can control the menu layout.

Figure 9-6 shows one example of using a menu. In this case we're choosing `Global Settings` from the `Edit` menu.

It's interesting to note these menu options are merely wrappers for Vim commands. In fact, that is exactly how you can create and customize your own menu entries, which we discuss shortly.

Figure 9-6. Cascading edit menu (Windows version)

If you pay attention to the menus, including the keystrokes or commands shown on the right side, you can learn Vim commands over time. For example, in Figure 9-6, although it's handy for beginners to find the familiar Undo command in the Edit menu, where it appears in other popular applications, it is *much* faster and easier to use the Vim u keystroke, which is shown in the menu.

As shown in Figure 9-6, each menu starts with a dashed line containing a picture of scissors. Clicking this line "tears off" the menu to create a freestanding window in which that submenu's options are available without going to the menu bar. If you clicked the dashed line above the `Toggle Pattern Highlight` menu option in Figure 9-6, you would see something like Figure 9-7. You can position the free-floating menu anywhere on your desktop.

Figure 9-7. Tearing off a menu

Now all of the commands on this submenu are immediately available with just one click in the submenu's window. Each menu selection is mapped to a button. If a menu selection itself is a submenu, it is represented by a button with greater-than signs (which look like rightward-pointing arrows) at the right side of the button. Clicking these arrows expands the submenu.

Basic menu customization

`gvim` stores menu definitions in a file named *$VIMRUNTIME/menu.vim*.

Defining menu items is similar to mapping. As you saw in the section "Using the map Command" on page 124, you can map a key like this:

```
:map <F12> :set syntax=html<CR>
```

Menus are handled very similarly.

Suppose that, rather than map F12 to set the syntax to `html`, we want a special "HTML" entry on our File menu to do this task. Use the `:amenu` command:

```
:amenu File.HTML :set filetype=html<CR>
```

The four characters `<CR>` are to be typed as shown and are part of the command.

Now look at your File menu. You should see a new HTML entry, as shown in Figure 9-8. By using `amenu` instead of `menu`, we ensure that the entry is available in all modes (command, insert, and normal).

Figure 9-8. HTML menu item under the File menu

The `menu` command adds the entry to the menu only in command mode; the entry does not appear in insert and normal modes.

The location for a menu entry is specified by a series of cascading menu names separated by periods (`.`). In our example, `File.HTML` added the menu entry "HTML" to the File menu. The last entry in the series is the one you want to add. Here we've added it to an existing menu, but we'll soon see that we can just as easily create a whole cascading series of new menus.

Be sure to test your new menu selection. For example, we started editing a file that Vim treats as an XML file, as can be seen in the status line in Figure 9-9 (see the section "A Nice Vim Piggybacking Trick" on page 296 for how to set up the status

line). We've customized the status line so that Vim and `gvim` display the currently active file type on the far right.

```
0x6F  line:1,  col:2 All [xml]
```

Figure 9-9. Status line showing XML file type before the new menu action

After invoking our new HTML menu item, the Vim status line verifies that the menu item worked and that the file type is now HTML—see Figure 9-10.

```
0x6F  line:1,  col:2 All [html]
```

Figure 9-10. Status line showing HTML file type after the new menu action

Notice that the HTML menu item we added doesn't have a shortcut or command on the righthand side. So let's redo the menu addition and include this nice enhancement.

First, delete the existing entry:

```
:aunmenu File.HTML
```

> If you add a menu entry for `ex` command mode using only the `menu` command, you can remove it using `unmenu`.

Next, add a new HTML menu item that displays the command you associated to the item:

```
:amenu File.HTML<TAB>filetype=html<CR> :set filetype=html<CR>
```

The specification of the menu entry is now followed by <TAB> (typed literally) and `filetype=html<CR>`. In general, to display text on the righthand side of the menu, place it after the string <TAB> and terminate it with <CR>. Figure 9-11 shows the resulting File menu.

Figure 9-11. HTML menu item displaying associated command

> If you want spaces in the descriptive text of the menu item (or in the menu name itself), quote the spaces with backslashes (\). If you don't, Vim uses everything after the first space character for the definition of the menu action. In the previous example, if we wanted `:set filetype=html` instead of just `filetype=html` for the descriptive text, the `:amenu` command would have to be:
>
> `:amenu File.HTML<TAB>set\ filetype=html<CR> :set filetype=html<CR>`

In most cases, it's probably best not to modify the default menu definitions but instead to create separate, independent entries. This requires defining a new menu at the root level, but this is just as simple as adding an entry to an existing menu.

Continuing our example, let's create a new menu tree called `MyMenu` on the menu bar, and then add an HTML menu item to it. First, remove the HTML item from the File menu:

`:aunmenu File.HTML`

Next, enter the command:

`:amenu MyMenu.HTML<TAB>filetype=html :set filetype=html<CR>`

Figure 9-12 shows how your menu bar may appear.

Figure 9-12. Menu bar with "MyMenu" menu added

The menu commands offer more subtle control over where the menus appear and over their behavior, such as whether the command indicates any activity, or even whether the menu item is visible. We discuss these possibilities further in the following section.

More menu customization

Now that we see how easy it is to modify and extend gvim's menus, let's look at more examples of customization and control.

Our previous example didn't specify where to put the new MyMenu menu, and gvim arbitrarily placed it on the menu bar between Window and Help. gvim lets us control the position with its notion of *priority*, which is simply a numerical value assigned to each menu to determine where it goes on the menu bar. The higher this value is, the further to the right the menu appears. Unfortunately, the way users think of priority is the opposite of how it's defined by gvim. To get priority straight, look back at the order of menus in Figure 9-5 and compare it to gvim's default menu priorities, as listed in Table 9-1.

Table 9-1. gvim's default menu priorities

Menu	Priority
File	10
Edit	20
Tools	40
Syntax	50
Buffers	60
Window	70
Help	9999

Most users would consider File a higher priority than Help (which is why File is on the left and Help is on the right), but the priority of Help is higher. So just think of the priority value as an indication of how far to the right a menu appears.

You can define a menu's priority by prepending its numeric value to the menu command. If no value is specified, a default value of 500 is assigned, which explains why MyMenu ended up where it did in our earlier example: it landed between Window (priority 70) and Help (priority 9999).

Assume we want our new menu to be between the File and Edit menus. We need to assign MyMenu a numeric priority greater than 10 and less than 20. The following command assigns a priority of 15, leading to the desired effect:

```
:15amenu MyMenu.HTML<TAB>filetype=html :set filetype=html<CR>
```

> Once a menu exists, its position is fixed for an entire editing session and does not change in response to additional commands that affect the menu. For example, you cannot change a menu's position by adding a new item to it and prefixing the command with a different priority value.

To add some more confusion to priorities and menu placement, you can also control item placement *within* a menu by specifying a priority. Higher-priority menu items appear further down in the menu than lower-priority items, but the syntax is different from priority definitions for menus.

We'll extend one of our earlier menu examples by assigning a very high value (9999) to the HTML menu item, so that it appears at the bottom of the File menu:

```
:amenu File.HTML .9999 <TAB>filetype=html<CR> :set filetype=html<CR>
```

Why is there a period before 9999? You need to specify two priorities here, separated by a period: one for File and one for HTML. We are leaving the File priority blank because it's a preexisting menu and can't be changed.

In general, priorities for a menu item appear between the item's menu placement and the item's definition. For every level in the menu hierarchy, you must specify a priority or else include a period to indicate that you're leaving it blank. Thus, if you add an item deep in the menu hierarchy—such as under Edit → Global Settings → Context lines → Display—and you want to assign the priority 30 to the last item (Display), you would specify the priority as …30, and the placement together with the priority would look like:

```
Edit.Global\ Settings.Context\ lines.Display ...30
```

As with menu priorities, menu item priorities are fixed once they are assigned.

Finally, you can control menu "whitespace" with gvim's menu separators. Use the same definition as you would to add a menu item, but instead of a plain command name, place a hyphen (-) before and after it. See the line in the next example with the identifiers 2 and 3.

Putting it all together

Now we know how to create, place, and customize menus. Let's make our example a permanent part of our gvim environment by adding the commands we discussed to the *.gvimrc* file. The sequence of lines should look something like:

```
" add XML/HTML/XHTML menu between File and Edit menus
❶15amenu MyMenu.XML<TAB>filetype=xml :set filetype=xml<CR>
❷amenu ❸.600 MyMenu.-Sep- :
❹amenu ❺.650 MyMenu.HTML<TAB>filetype=html :set filetype=html<CR>
❻amenu ❼.700 MyMenu.XHTML<TAB>filetype=xhtml :set filetype=xhtml<CR>
```

We now have a top-level, personalized menu with three favorite file type commands quickly available to us. There are a few important things to note in this example:

- The first command (❶) uses the prefix 15, telling gvim to use priority 15. For an uncustomized environment, this places the new menu between the File and Edit menus.

- The subsequent commands (❷, ❹, and ❻) do *not* specify the priority, because once a priority is determined, no other values are used.

- We've used the submenu priority syntax (❸, ❺, and ❼) after the first command to ensure the correct order for each new item. Notice we started with the first definition of .600. This assures that the submenu item is placed behind the first one we defined, because we didn't assign *that* priority and it therefore defaulted to 500.

For even handier access, click on the "scissors" tear-off line to have your personalized floating menu, as shown in Figure 9-13.

Figure 9-13. Personalized floating tear-off menu

Toolbars

Toolbars are long strips of icons that allow quick access to program functions. On GNU/Linx, for instance, gvim displays the toolbar shown in Figure 9-14 at the top of the window.

Figure 9-14. gvim's toolbar (Linux version)

Table 9-2 shows the toolbar icons and their meanings.

Table 9-2. `gvim` toolbar icons and their meanings

Icon	Description	Icon	Description
	Open file dialog		Find next occurrence of search pattern
	Save current file		Find previous occurrence of search pattern
	Save all files		Choose saved edit session to load
	Print buffer		Save current edit session
	Undo last change		Choose Vim script to run
	Redo last action		Make the current project with the `make` command
	Cut selection to clipboard		Build tags for the current directory tree
	Copy selection to clipboard		Jump to tag under cursor
	Paste clipboard into buffer		Open help
	Find and replace		Search help

If these icons are not familiar or intuitive, you can make the toolbar show both text and icons. Issue this command:

```
:set toolbar="text,icons"
```

As with many advanced features, Vim requires toolbar features to be turned on during compilation so people who don't want them can save memory by not including them. The toolbar does not exist unless one of the +GUI_GTK, +GUI_Athena, +GUI_Motif, or +GUI_Photon features is compiled into your version of `gvim`. Appendix D, "vi and Vim: Source Code and Building", explains how to recompile Vim, during which the link to the `gvim` executable is created.

We modify the toolbar very much like we do menus. As a matter of fact, we use the same :menu command, but with extra syntax to specify graphics. Although an algorithm exists to help gvim find the icon associated with each command, we recommend explicitly specifying the icon graphic.

gvim treats the toolbar as a one-dimensional menu. And just as you control the right-to-left position of new menus, you can control the position of new toolbar entries by prefixing the menu command with a number that determines its positional *priority*. Unlike menus, there is no notion of creating a new toolbar. All new toolbar definitions appear on the single toolbar. The syntax for adding a toolbar selection is:

```
:amenu icon=/some/icon/image.bmp ToolBar.NewToolBarSelection Action
```

where */some/icon/image.bmp* is the path of the file containing the toolbar button or image (usually an icon) to display in the toolbar, *NewToolBarSelection* is the new entry for the toolbar button, and *Action* defines what the button does.

For example, let's define a new toolbar selection that, when clicked or selected, brings up a DOS window in Windows. Assuming the Windows path is set up correctly, we will define our toolbar selection to start a DOS window from within gvim by executing the following (this is its *Action*):

```
:!cmd
```

For the new selection's toolbar button, or image, we use an icon showing a DOS command prompt, shown in Figure 9-15, which on our system is stored in *$HOME/dos.bmp*.

Figure 9-15. DOS icon

Execute the command:

```
:amenu icon="c:$HOME/dos.bmp" ToolBar.DOSWindow :!cmd<CR>
```

This creates a toolbar entry and adds our icon at the end of the toolbar. The toolbar should now look like Figure 9-16. The new icon appears on the rightmost end of the toolbar.

Figure 9-16. Toolbar with added DOS command button

Tooltips

gvim lets you define tooltips for both menu entries and toolbar icons. Menu tooltips display in the gvim command-line area when the mouse is over that menu selection. Toolbar tooltips pop up graphically when the mouse hovers over a toolbar icon. For example, Figure 9-17 shows the tooltip that pops up when we put the mouse over the toolbar's Find Previous button.

Figure 9-17. Tooltip for the Find Previous icon

The :tmenu command defines tooltips for both menus and toolbar items. The syntax is:

```
:tmenu TopMenu.NextLevelMenu.MenuItem tool tip text
```

where *TopMenu.NextLevellMenu.MenuItem* defines the menu as it cascades from the top level all the way to the menu item for which you wish to define a tooltip. So, for example, a tooltip for the Open menu item under the File menu would be defined with the following command:

```
:tmenu File.Open Open a file
```

Use ToolBar for the top-level "menu" if you are defining a toolbar item (there is no real top-level menu for a toolbar).

Let's define a pop-up tooltip for the DOS toolbar icon we created in the previous section. Enter the command:

```
:tmenu ToolBar.DOSWindow Open up a DOS window
```

Now when you hover over the newly added toolbar icon, you can see the tooltip, as shown in Figure 9-18.

Figure 9-18. Toolbar with added DOS command and its new tooltip

gvim in Microsoft Windows

gvim is increasingly popular among MS-Windows users. Veteran vi and Vim users will find the Windows version excellent, and it is probably the most current version across all operating systems.

The self-installing executable should automatically and seamlessly integrate Vim into the Windows environment. If it doesn't, consult the *gui-w32.txt* help file in the Vim runtime directory for `regedit` instructions. Because this involves editing the Windows Registry, do *not* try it if it's a procedure with which you are the slightest bit uncomfortable. You may be able to find someone with more expertise to help you. It is a common but nontrivial exercise.

Longtime Windows users are familiar with the *clipboard*, a storage area where text and other information is kept to facilitate copy, cut, and paste operations. Vim supports interaction with the Windows clipboard. Simply highlight text in visual mode and click the Copy or Cut menu item to store Vim text in the Windows clipboard. You can then paste that text into other Windows applications.

gvim in the X Window System

Users familiar with the X environment can define and use many of the tunable X features. For example, you can define many resources with the standard class definitions typically defined in the *.Xdefaults* file.

Note that these standard X resources are useful only for the Motif or Athena versions of the GUI. Obviously, the Windows version has no understanding of X resources. Not so obviously, X resources are not picked up by KDE or Gnome either, and on a modern system, `gvim` is likely to be based on one of those two toolkits.

Running gvim in Microsoft Windows WSL

As of this writing, Microsoft has released two major versions of its virtualized support for GNU/Linux distributions, *Windows Subsystem for Linux* (WSL). They are commonly referred to as WSL and WSL 2.

WSL provides compatible interfaces that allow GNU applications to run and thus enables full GNU/Linux distributions to run under Windows. WSL 2 ups the ante and provides a running native Linux kernel executing in a virtual environment. The deep details are beyond this book's scope, but it bears mentioning that one of us has successfully used WSL often and productively and attests to Vim's full functionality when executed in Microsoft's `Terminal` application (it's a console application). While this was a pleasant surprise, it was even more exciting to learn that Microsoft is adding more GUI support for WSL Linux.

This section will help you run `gvim` from WSL 2, displayed in native Windows. For those familiar with X11, the approach is fairly standard, but there are some

configuration tweaks necessary for Windows, and an appropriate `gvim` package must be installed for the target GNU/Linux distribution.

Installing gvim in WSL 2

We describe the installation and configuration of `gvim` in a WSL 2 Ubuntu distribution.

First, to verify whether `gvim` exists, use `dpkg` to search for the `gvim` binary. To eliminate false positive results (man pages, configuration files, etc.), search for "bin/gvim":

```
$ dpkg -S bin/gvim
vim-gui-common: /usr/bin/gvimtutor
```

Nope, no `gvim`! `gvimtutor` is not `gvim` and simply drops back to terminal-based Vim when `gvim` is not installed.

Now, as root (via `sudo`), install the `gvim` package. While we happen to know that the package is `vim-gtk3`, it's sometimes useful to let the Ubuntu package manager `apt` provide hints. Ask `apt` simply to install `gvim`, and you see three possibilities, the `vim-gtk3` package being our choice:

```
vim@office-win10:~$ sudo apt install gvim
[sudo] password for vim:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Package gvim is a virtual package provided by:
  vim-gtk 2:8.0.1453-1ubuntu1.4
  vim-athena 2:8.0.1453-1ubuntu1.4
  vim-gtk3 2:8.0.1453-1ubuntu1.4
You should explicitly select one to install.
```

Now we know the package we want and install that package with `apt`:

```
vim@office-win10:~$ sudo apt install vim-gtk3
...
```

All things being equal (we can't go through all the possibilities of failed installations, but this should be reliable), `gvim` should now be available in your Linux subsystem. Accordingly, the installation updates your PATH and rehashes `gvim` into your available commands. Verify this with the `type` command:

```
$ type gvim
gvim is /usr/bin/gvim
```

Okay, we're almost there. You can execute `gvim`, but it's still not quite what we want. If you execute `gvim`, you'll see a response like:

```
$ gvim
E233: cannot open display
Press ENTER or type command to continue
```

and when you press $\boxed{\text{ENTER}}$ to continue, `gvim` falls back to the text-based terminal Vim. We need to complete X Window setup by establishing an X Windows server in the Windows environment and asking the Linux `gvim` to display graphically to it.[3]

Installing an X Server for Windows

As just mentioned, we must establish a server from Microsoft Windows to receive graphical requests, enabling our WSL Linux instance of `gvim` to display transparently on the Windows Desktop. In our example we will use the freely available Windows server "Xming."

Download the latest Xming server installer and execute it. The installer splash banner can be seen in Figure 9-19.

Figure 9-19. Xming installer

Configuring the X Server for Windows

Xming installs two executables:

- "XLaunch," a configuration wizard simplifying common instances of Xming
- "Xming," the actual X server

---

3 This can be a little confusing. Bear with us. Windows and X Windows *are not the same*, but both are important here. Windows is Windows, your familiar Microsoft Desktop. X Windows is a graphical server running in your Microsoft Windows and knows how to display graphics from remote systems.

Open the Windows Applications menus and find the Xming folder. You will see something like Figure 9-20.

Figure 9-20. Xming apps installed from Xming Setup

Execute XLaunch to configure Xming. XLaunch guides you through standard X-ish stuff. Figure 9-21 is XLaunch's first screen. Select the same options as you see in these figures.

Figure 9-21. XLaunch opening dialog, "Display settings"; we select "Multiple windows"

Select "Multiple windows," and fill in Display number with the number zero, 0.[4]

The next dialog defines the X "Session type," as represented in Figure 9-22.

---

4 The explanation of the various options are out of scope for this book. Briefly, we choose "Multiple windows" because that option lets gvim display graphically in a Windows window.

Figure 9-22. XLaunch dialog, "Session type"; we select "Start no client"

Select "Start no client." This simply tells XLaunch to start the Xming X Window server by itself. In that role, Xming will wait and display applications requesting display from the remote host, which in this case is our WSL Linux host.

Next, XLaunch records other miscellaneous common X Windows parameters. See Figure 9-23.

Figure 9-23. XLaunch dialog, "Additional parameters"; we select "Clipboard" and "No Access Control"

Select "Clipboard" and "No Access Control." Note that the selection "No Access Control" is not preselected.

> We select the option "No Access Control" for the convenience of not dealing with X Windows security mechanisms. We do this under the assumption that the computer is in a "safe" environment, e.g., a home network wherein no attacks would occur against the Xming X server. This would *not* be the proper choice in any public network or business office setting.

We have now configured Xming and are ready to launch. The final XLaunch dialog in Figure 9-24 shows options to save your new configuration (we won't) and start Xming.

Figure 9-24. XLaunch "Finish configuration" dialog

Once you've started Xming, verify it is running and ready to display Ubuntu X Windows applications by looking for the Xming icon in the system tray (see Figure 9-25).

Figure 9-25. Xming icon in the Windows system tray

So we have all the working pieces to start our Ubuntu instance of `gvim`, but we're not *quite* done. If you execute `gvim`, it still displays an error message and degrades to the terminal Vim.

This is because we need to tell our Ubuntu system *where* to display `gvim`. In this case, we must tell Ubuntu to point to our Microsoft Windows Desktop. While it's painfully obvious to us *prima facie*, X Windows applications by definition must request an X server to display their content, which we have not done.

We define in Ubuntu *where* we want to display `gvim` with our Microsoft Windows network address *and* its associated X server display.[5] The format is *hostname:DIS-PLAY*, where *hostname* will be our Windows IP address, and *DISPLAY* will be `0`.

The easiest way to find the Microsoft Windows IP address is to look in the configuration file `/etc/resolv.conf` in Ubuntu, where Ubuntu's "nameserver" is the Microsoft Windows address:

```
$ cat /etc/resolv.conf
# This file was automatically generated by WSL. To stop automatic generation of
# this file, add the following entry to /etc/wsl.conf:
# [network]
# generateResolvConf = false
nameserver 172.17.224.1
```

or:

```
$ grep nameserver /etc/resolv.conf
nameserver 172.17.224.1
```

We see in our case that Ubuntu's nameserver is 172.17.224.1. So the definition for our X server will be `172.17.224.1:0.0`.[6]

There are multiple ways to convey to `gvim` the display on which to present. The most common way is by the environment variable `DISPLAY`, which we set with the shell `export` command:

```
$ export DISPLAY=172.17.224.1:0.0
```

Now we're ready. Start `gvim`, and you should now see a GUI version running as a window in your Desktop. See Figure 9-26. Note that this is real: we've captured a moment in time of authoring this book with `gvim` editing *this* chapter. You'll notice underneath the terminal and command line where we edited the real chapter file. Cool. ☺

---

5 Yes, your MS-Windows computer now has a network interface visible to the Linux subsystem. This is how WSL works, as it is a virtual computer with its own network address, thus requiring real network semantics so Linux and MS-Windows can talk to each other.

6 Note the `0.0` form for the display number. This is related to the display and possible multiple screens. For most use cases (simple ones), the display should be `0.0`.

Figure 9-26. Ubuntu `gvim` *displayed in Microsoft Windows Desktop*

Notice the numbered callouts:

❶ This is the actual `gvim` window running transparently as an application on the Microsoft Windows Desktop.

❷ This is the underlying Microsoft Terminal in which we are running a WSL instance of Ubuntu. You can see the command lines executed that we just mentioned in the text.

❸ This is `xeyes`, a frivolous but popular little X Windows application, watching us as we work.[7]

We've shown you now how to configure and use `gvim` from a Microsoft Windows WSL Linux instance and display that session transparently. There are myriad ways to set up and configure X Windows servers and clients. What you've learned here is a big step toward using `gvim` *and* understanding X Windows fundamentals. A full discussion of X and how you configure and customize it has been exhaustively documented elsewhere and is beyond the scope of this book. For a brief (or not so brief) introduction to X, we suggest the X man page.

---

7 You may not realize it, but this entire exercise also enables any and all X Windows applications available in the Ubuntu instance, `xeyes` being one of many. Once you have set up the X server and defined the network and `DISPLAY`, all X Windows applications use the same display. Congratulations! You've just completed a useful lesson in X Windows. To verify further, try executing the X Windows terminal application, `xterm`.

GUI Options and Command Synopsis

Table 9-3 summarizes the commands and options specifically associated with `gvim`. These are added to Vim when it is compiled with GUI support, and they take effect when it is invoked as `gvim` or `vim -g`.

Table 9-3. `gvim`-specific options

Command, option, or flag	Type	Description
`guicursor`	Option	Settings for cursor shape and blinking
`guifont`	Option	Names of single-byte fonts to be used
`guifontset`	Option	Names of multibyte fonts to be used
`guifontwide`	Option	List of font names for double-wide characters
`guiheadroom`	Option	Number of pixels to leave for window decorations
`guioptions`	Option	Which components and options are used
`guipty`	Option	Use a pseudo-tty for ":!" commands
`guitablabel`	Option	Custom label for a tab page
`guitabtooltip`	Option	Custom tooltip for a tab page
`toolbar`	Option	Items to show in the toolbar
`-g`	Flag	Start the GUI (which also allows the other options)
`-U` *gvimrc*	Flag	Use `gvim` startup file, named *gvimrc* or something similar, when starting the GUI
`:gui`	Command	Start the GUI (on Unix-like systems only)
`:gui` *filename…*	Command	Start the GUI and edit the specified files
`:menu`	Command	List all menus
`:menu` *menupath*	Command	List menus starting with *menupath*
`:menu` *menupath action*	Command	Add menu *menupath*, to perform *action*
`:menu` *n menupath action*	Command	Add menu *menupath* with positional priority of *n*
`:menu ToolBar.`*toolbarname action*	Command	Add toolbar item *toolbarname* to perform *action*
`:tmenu` *menupath text*	Command	Create tooltip for menu item *menupath* with text of *text*
`:unmenu` *menupath*	Command	Remove menu *menupath*

Multiple Windows in Vim

By default, Vim edits all its files in a single window, showing just one buffer at a time as you move between files or to different parts of a single file. But Vim also offers multiwindow editing, which can make complex editing tasks easier. This is different from starting multiple instances of Vim on a graphical terminal. This chapter covers the use of multiple windows in a single instance of a running Vim process (which we'll call a *session*).

You can initiate your editing session with multiple windows or create new windows after a session starts. You can add windows to your editing session up to the limit imposed by sanity, and you can delete them back to a single window.

Multiple windows today makes more sense than ever with high-resolution monitors being the norm. At the time of the seventh edition of this book, WXGA (1280x800) was considered decent resolution. Today (late 2021), for about the same price, it's easy to find monitors in 4K resolution (Ultra HD: 3840x2160) for around $400. That's around *nine times* the resolution!

Then Vim's multiple-windows feature enhanced users' editing by offering multiple viewports and glimpses into a single file or multiple files simultaneously. This was a giant step forward for powerful editing, but it was often at the cost of compromising real estate by either setting line wrap parameters so entire lines remained visible or setting line shift options to scroll left and right as lines were clipped at the sides of windows.

Now with high-resolution screens, Vim's multiple windows provide the same powerful features, and users can easily split windows side-by-side and still get full-width text displays for each window.[1]

Here are some examples of when multiple windows make your life easier:

- Editing a number of files that need to be formatted the same way, where you would like to compare them visually as you go along
- Cutting and pasting text quickly and repeatedly among multiple files or multiple parts of a single file
- Displaying one part of a file for reference, to facilitate work elsewhere in the same file
- Comparing two versions of a file

Vim offers many window-managing convenience features, including the ability to:

- Split windows horizontally or vertically
- Navigate from one window to another and back again quickly
- Copy and move text to and from multiple windows
- Move and reposition windows
- Work with buffers, including hidden buffers (to be described later)
- Use external tools such as the `diff` command with multiple windows

In this chapter, we guide you through the multiwindow experience. We show you how to start a multiwindow session, discuss features and tips for the editing session, and describe how to exit your work and ensure that all your work is properly saved (or abandoned, if you wish!). The following topics are covered:

- Initializing or starting multiwindow editing
- Multiwindow `:ex` commands
- Moving the cursor from window to window
- Moving windows around the display
- Resizing windows
- Buffers and their interaction with windows
- Playing tag with windows

---

1 Of course, as described in this chapter, you still have the option to slice and dice windows to very tiny sizes for whatever fills your editing needs.

- Tabbed editing (like the tabs offered by modern internet browsers and dialog boxes)
- Closing and quitting windows

Initiating Multiwindow Editing

You can initiate multiwindow editing when you start Vim, or you can split windows within your editing session. Multiwindow editing is dynamic, allowing you to open, close, and navigate among multiple windows at any point, from most contexts.

Multiwindow Initiation from the Command Line

By default, Vim opens only one window for a session, even if you specify more than one file. While we don't know for sure why Vim would not open multiple windows for multiple files, it may be because using just a single window is consistent with vi. Multiple files occupy multiple buffers, with each file in its own buffer. (Buffers are discussed shortly.)

To open multiple windows from the command line, use Vim's -o option. For example:

```
$ gvim -o file1.txt file2.txt
```

This opens the editing session with the display horizontally split into two equal-sized windows, one for each file (see Figure 10-1). For each file named on the command line, Vim tries to open a window for editing. If Vim cannot split the screen into enough windows for the files, the first files listed in the command-line arguments get windows, while the remaining files are loaded into buffers not visible (but still available) to you.

Figure 10-1. Results of gvim -o file1.txt file2.txt *(Linux* gvim*)*

Another form of the command line preallocates the windows by appending a number *n* to `-o`:

```
$ gvim -o5 file1.txt file2.txt
```

This opens a session with the display horizontally split into five equal-sized windows, the topmost of which contains *file1.txt* and the second of which contains *file2.txt* (see Figure 10-2).

Figure 10-2. Results of `gvim -o5 file1.txt file2.txt` *(Linux* `gvim`*)*

> When Vim creates more than one window, its default behavior is to create a status line for each window (whereas the default behavior for a single window is not to display any status line). You can control this behavior with Vim's `laststatus` option, e.g.:
>
> ```
> :set laststatus=1
> ```
>
> Set `laststatus` to 2 to always see a status line for each window, even in single window mode. It is best to set this in your *.vimrc* file.

Because window size affects readability and usability, you may want to control Vim's limits for window sizes. Use Vim's `winheight` and `winwidth` options to define reasonable limits for the current window (other windows may be resized to accommodate it).

Multiwindow Editing Inside Vim

You can initiate and modify the window configuration from within Vim. Create a new window with the `:split` command. This breaks the current window in half, showing the same buffer in both halves. Now you can navigate independently in each window on the same file.

> There are convenience key sequences for many of the commands in this chapter. In this case, for instance, CTRL-W S splits a window. All Vim window-related commands begin with CTRL-W, with the "W" being mnemonic for "window." For the purposes of discussion, we show only the command-line methods because they provide the added power of optional parameters that customize the default behavior. If you find yourself using commands routinely, you can easily find the corresponding key sequence in the Vim documentation, as described in "Built-In Help" on page 165.

Similarly, you can create a new, vertically separated editing window with the `:vsplit` command (see Figure 10-3).

Figure 10-3. Vertically split window (Linux `gvim`)

For each of these methods, Vim splits the window (horizontally or vertically), and since no file was specified on the `:split` command line, you end up editing the same file with two views or windows.

> Don't believe you're editing the same file at the same time? Split the editing window and scroll each window so that each shows the same area of the file. Make changes. Watch the other window. Magic!

Why or how is this useful? One common use, when writing shell scripts or C programs, is to code a block of text that describes the program's usage. Typically, the program will display the block when passed a `--help` option. We split the display so that one window displays the usage text, and we use this as a template to edit the code in the other window that parses all the options and command-line arguments described in the usage text. Often (almost always) this code is complex and ends up far enough from the usage text that we can't display everything in a single window.

If you want to edit or browse another file without losing your place in your current file, provide the new file as an argument to your `:split` command. For instance:

```
:split otherfile
```

The next section describes splitting and unsplitting windows in more detail.

Opening Windows

This section goes into depth about how to get the precise behavior you want when you split your window.

New Windows

As discussed previously, the simplest way to open a new window is to issue `:split` (for a horizontal division) or `:vsplit` (for a vertical division). A more in-depth discussion of the many commands and variations follows. We also include a command synopsis for quick reference.

Options During Splits

The full `:split` command to open a new horizontal window is:

```
:[n]split [++opt] [+cmd] [file]
```

where:

n
    Tells Vim how many lines to display in the new window, which goes at the top.

opt
    Passes Vim option information to the new window session (note that it must be preceded by two plus signs).

cmd
    Passes a command for execution in the new window (note that it must be preceded by a single plus sign).

file
    Specifies a file to edit in the new window.

For example, suppose you are editing a file and want to split the window to edit another file named *otherfile*. You want to ensure that the session uses a `fileformat` of `unix` (which ensures the use of a line feed to end each line instead of a carriage return and line feed combination). Finally, you want the window to be 15 lines tall. Enter:

```
:15split ++fileformat=unix otherfile
```

To simply split the screen, showing the same file in both windows and using all the current defaults, you can use the `vi` mode commands `CTRL-W` `S`, `CTRL-W` `SHIFT-S`, or `CTRL-W` `CTRL-S`.

> If you want windows to always split equally, set the `equalalways` option, preferably putting it in your *.vimrc* to make it persistent over sessions. By default, setting `equalalways` splits both horizontal and vertical windows equally. Add the `eadirection` option (`hor`, `ver`, `both`, for horizontal, vertical, or both, respectively) to control which direction splits equally.

The following form of the `:split` command opens a new horizontal window as before, but with a slight nuance:

```
:[n]new [++opt] [+cmd] [file]
```

In addition to creating the new window, the `WinLeave`, `WinEnter`, `BufLeave`, and `BufEnter` autocommands execute. (For more information on autocommands, see the section "Autocommands" on page 300.)

Along with the horizontal split commands, Vim offers analogous vertical ones. So, for example, to split a vertical window, instead of `:split` or `:new`, use `:vsplit` and `:vnew`, respectively. The same optional parameters are available as for the horizontal split commands.

There are two horizontal split commands without vertical cousins:

`:sview` *filename*

Splits the screen horizontally to open a new window and sets the `readonly` option for that buffer. `:sview` requires the filename argument.

`:sfind` [++ *opt*] [+ *cmd*] *filename*

Works like `:split`, but looks for the *filename* in the `path`. If Vim does not find the file, it doesn't split the window.

Conditional Split Commands

Vim lets you specify a command that causes a window to open if a new file is found. `:topleft` *cmd* tells Vim to execute *cmd* and display a new window with the cursor at the top left if *cmd* opens a new file. The command can produce three different results:

- *cmd* splits the window horizontally, and the new window spans the top of the Vim window.
- *cmd* splits the window vertically, and the new window spans the left side of the Vim window.
- *cmd* causes no split but instead positions the cursor at the top left of the current window.

In addition to the conditional split command `:topleft`, Vim offers analogous commands `:vertical`, `:leftabove` and `:aboveleft`, `:rightbelow` and `:belowright`, and `:botright`. You can find detailed descriptions on their use with Vim's `:help` command.

Window Command Summary

Table 10-1 summarizes the commands for splitting windows.

Table 10-1. Summary of window commands

ex command	vi command	Description
`:[`*n*`]split [++`*opt*`] [+`*cmd*`] [`*file*`]`	CTRL-W S CTRL-W SHIFT-S CTRL-W CTRL-S	Split the current window in two from side to side, placing the cursor in the new window. The optional *file* argument places that file in the newly created window. The windows are created as equal in size as possible, determined by free window space.
`:[`*n*`]new [++`*opt*`] [+`*cmd*`]`	CTRL-W N CTRL-W CTRL-N	Same as `:split`, but start the new window editing an empty file. Note that the buffer will have no name until one is assigned.
`:[`*n*`]sview [++`*opt*`] [+`*cmd*`] [`*file*`]`		Read-only version of `:split`.
`:[`*n*`]sfind [++`*opt*`] [+`*cmd*`] [`*file*`]`		Split the window and open *file* (if specified) in the new window. Look for *file* in the `path`.
`:[`*n*`]vsplit [++`*opt*`] [+`*cmd*`] [`*file*`]`	CTRL-W V CTRL-W CTRL-V	Split the current window in two from top to bottom and open *file* (if specified) in the new window.
`:[`*n*`]vnew [++`*opt*`] [+`*cmd*`]`		Vertical version of `:new`.

Moving Around Windows (Getting Your Cursor from Here to There)

It's easy to move from window to window with a mouse in both `gvim` and Vim. `gvim` supports clicking with the mouse by default, whereas in Vim you can enable the behavior with the `mouse` option. A good default setting for Vim is `:set mouse=a`, which activates the mouse for all uses: command line, input, and navigation.

If you don't have a mouse, or if you prefer to control your session from the keyboard, Vim provides a full set of navigation commands to move quickly and accurately among session windows. Happily, Vim uses the mnemonic prefix keystroke CTRL-W consistently for window navigation. The keystroke that follows defines the motion or other action, and the window navigation commands should be familiar to experienced `vi` and Vim users because they map closely to the same motion commands for editing.

Rather than describe each command and its behavior, we will consider an example. The command-synopsis table should then be self-explanatory.

To move from the current Vim window to the next one, type CTRL-W J (or CTRL-W ↓ or CTRL-W CTRL-J). The CTRL-W is the mnemonic for "window" command, and the `j` is analogous to Vim's `j` command, which moves the cursor to the next line.

Table 10-2 summarizes the window navigation commands.

As with many Vim and `vi` commands, these can be multiply executed by prefixing them with a count. For example, 3 CTRL-W J tells Vim to jump to the third window down from the current window.

Table 10-2. Window navigation commands

Command	Description
CTRL-W W CTRL-W CTRL-W	Move to the next window below or to the right. Note that this command, unlike CTRL-W J, cycles through all of the Vim windows. When the lowermost window is reached, Vim restarts the cycle and moves to the top leftmost window.
CTRL-W ↓ CTRL-W CTRL-J CTRL-W J	Move to the next window down. Note that this command does not cycle through the windows; it simply moves down to the next window below the current window. If the cursor is in a window at the bottom of the screen, this command has no effect. Also, this command bypasses adjacent windows on its "way down"; for example, if there is a window to the right of the current window, the command does not jump across to the adjacent window. (Use CTRL-W CTRL-W to cycle through windows.)
CTRL-W ↑ CTRL-W CTRL-K CTRL-W K	Move to the next window up. This is the opposite-direction counterpart to the CTRL-W J command.

Command	Description
CTRL-W ← CTRL-W H CTRL-W BACKSPACE	Move to the window to the left of the current window.
CTRL-W → CTRL-W CTRL-L CTRL-W L	Move to the window to the right of the current window.
CTRL-W SHIFT-W	Move to the next window above or to the left. This is the upward counterpart to the CTRL-W W command (note the difference in case).
CTRL-W T CTRL-W CTRL-T	Move to the top leftmost window.
CTRL-W B CTRL-W CTRL-B	Move to the bottom rightmost window.
CTRL-W P CTRL-W CTRL-P	Move to the previous (last accessed) window.

Mnemonic Tips

The letters t and b are mnemonic for *top* and *bottom* windows.

In keeping with the convention that lowercase and uppercase implement opposites, CTRL-W W moves you through the windows in the opposite direction from CTRL-W SHIFT-W.

The control characters do not distinguish between uppercase and lowercase; in other words, pressing SHIFT while pressing a CTRL- key itself has no effect. However, an upper/lowercase distinction *is* recognized for the regular keyboard key you press afterward.

Moving Windows Around

You can move windows two ways in Vim. One way simply swaps the windows on the screen. The other way changes the actual window layouts. In the first case, window sizes remain constant while windows change position on the screen. In the second case, windows not only move but are resized to fill the position to which they've moved.

Moving Windows (Rotate or Exchange)

Three commands move windows without modifying layout. Two of these rotate the windows positionally in one direction (to the right or down) or the other (to the left or up), and the other one exchanges the position of two possibly nonadjacent windows. These commands operate *only* on the row or column in which the current window lives.

CTRL-W R rotates windows to the right or down. Its complement is CTRL-W SHIFT-R , which rotates windows in the opposite direction.

An easier way to imagine how these work is to think of a row or column of Vim windows as a one-dimensional array. CTRL-W R shifts each element of the array one position to the right and moves the last element into the vacated first position. CTRL-W SHIFT-R simply moves the elements in the other direction.

If there are no windows in a column or row that aligns with the current window, this command does nothing.

After Vim rotates the windows, the cursor remains in the window from which the rotate command executed; thus, the cursor moves with the window.

CTRL-W X and CTRL-W CTRL-X let you exchange two windows in a row or column of windows. By default, Vim exchanges the current window with the next window, and if there is no next window, Vim tries to exchange with the previous window. You can exchange with the *n*th next window by prepending a count before the command. For example, to switch the current window with the third next window, use the command 3 CTRL-W X .

As with the two previous commands, the cursor stays in the window from which the exchange command executes.

Moving Windows and Changing Their Layout

Five commands move and re-layout the windows: two move the current window to a full-width top or bottom window, two move the current window to a full-height left or right window, and the fifth moves the current window to another existing tab. See the section "Tabbed Editing" on page 233 for information on tabbed editing. The first four commands bear familiar mnemonic relationships to other Vim commands; for instance, CTRL-W SHIFT-K maps to the traditional notion of k as "up." Table 10-3 summarizes these commands.[2]

Table 10-3. Commands to move and re-layout windows

Command	Description
CTRL-W SHIFT-K	Move the current window to the top of the screen, using the full width of the screen.
CTRL-W SHIFT-J	Move the current window to the bottom of the screen, using the full width of the screen.
CTRL-W SHIFT-H	Move the current window to the left of the screen, using the full height of the screen.
CTRL-W SHIFT-L	Move the current window to the right of the screen, using the full height of the screen.
CTRL-W SHIFT-T	Move the current window to a new tab.

---

2 The shifted or capitalized letters here are a sort of amplification of managing windows. Remember that with these commands you are moving windows, not your cursor.

It is difficult to describe the exact behavior of these layout commands. After the move and expansion of the window to the full height or width of the screen, Vim redoes the window layout in a reasonable way. The behavior of the layout can also be influenced by some of the windows' options settings.

Window Move Commands: Synopsis

Tables 10-4 and 10-5 summarize the commands introduced in this section.

Table 10-4. Commands to rotate window positions

Command	Description
CTRL-W R CTRL-W CTRL-R	Rotate windows down or to the right.
CTRL-W SHIFT-R	Rotate windows up or to the left.
CTRL-W X CTRL-W CTRL-X	Swap positions with the next window, or if issued with a count *n*, swap with *n*th next window.

Table 10-5. Commands to change position and layout

Command	Description
CTRL-W SHIFT-K	Move the current window to the top of the screen and use the full width. The cursor stays with the moved window.
CTRL-W SHIFT-J	Move the current window to the bottom of the screen and use the full width. The cursor stays with the moved window.
CTRL-W SHIFT-H	Move the current window to the left of the screen and use the full height. The cursor stays with the moved window.
CTRL-W SHIFT-L	Move the current window to the right of the screen and use the full height. The cursor stays with the moved window.
CTRL-W SHIFT-T	Move the current window to a new tab. The cursor stays with the moved window. If the current window is the only window in the current tab, no action is taken.

Resizing Windows

Now that you're more familiar with Vim's multiwindowing features, you need a little more control over them. This section addresses how you can change the size of the current window, with, of course, effects on other windows in the screen. Vim provides options to control window sizes and window sizing behavior when opening new windows with split commands.

If you'd rather control window sizes *sans* commands, use `gvim` and let the mouse do the work for you. Simply click and drag window boundaries with the mouse to resize. For vertically separated windows, click the mouse on the vertical separator of pipe (|) characters. Horizontal windows are separated by their status lines.

Window Resize Commands

As you'd expect, Vim has vertical and horizontal resize commands. Like the other window commands, these all begin with CTRL-W and map nicely to mnemonic devices, making them easy to learn and remember.

CTRL-W = tries to resize all windows to equal size. This is also influenced by the current values of `winheight` and `winwidth`, discussed in the following section. If the available screen real estate doesn't divide equally, Vim sizes the windows to be as close to equal as possible.

CTRL-W - decreases the current window height by one line. Vim also has an `ex` command that lets you decrease the window size explicitly. For example, the command `:resize -4` decreases the current window by four lines and gives those lines to the window below it.

> There is another mechanism that changes window size: the `lines` option. Normally Vim manages the `lines` value, and you can see (and) use the value by referencing `lines` with the `ex` command `set`:
>
> `:set lines`
>
> However, for `gvim`, you can change the size of the graphical window by setting `lines`. A side effect of all of this is that in a terminal with only one window, if you set `lines` to a value less than the number of lines in the `vim` buffer window, Vim resizes the drawing real estate to the smaller count. Accordingly, Vim increases the lines allocated to the `ex` command line by the number of lines "lost" to the window buffer. For example, if you set `lines` to 15 in your *.vimrc* file and start `vim` in a 30-line window, Vim allocates 15 lines as the edit buffer. The remaining lines are allocated to the status line (1 line) and the `ex` command buffer (14 lines), which could be confusing since the `ex` command buffer is normally only 1 line. To avoid this side effect, we recommend you use `:set lines=xx` in your *.gvimrc* configuration file only.[3]

CTRL-W + increases the current window size by one line. The `:resize +n` command increases the current window size by *n* lines. Once the window's maximum height is reached, further use of this command has no effect.

---

3 You *can* modify the height of the `ex` command space using the `cmdheight` option (not to be confused with the `cmdwinheight` option).

Please see the section "Resize Your Window" on page 340 for a nice pair of key remappings that make window resizing much easier.

`:resize` *n* sets the horizontal size of the current window to *n* lines. It sets an absolute size, in contrast to the previously described commands that make a relative change.

`z`*n* sets the current window height to *n* lines. Note that *n* is *not* optional! Omitting it results in the `vi`/Vim command `z`, which moves the cursor to the top of the screen.

CTRL-W `<` and CTRL-W `>` decrease and increase the window width, respectively. Think of the mnemonic device of "shift left" (`<<`) and "shift right" (`>>`) to associate these commands to their function.

Finally, CTRL-W `|` resizes the current window to the widest size possible (by default). You can also specify explicitly how to change the window width with `:vertical resize` *n*. The *n* defines the window's new width.

Window Sizing Options

Several Vim options influence the behavior of the resize commands described in the previous section:

`winheight` *and* `winwidth`
These define the minimal window height and width, respectively, when a window becomes active. For example, if the screen accommodates two equal-sized windows of 45 lines, the default Vim behavior is to split them equally. If you were to set `winheight` to a value larger than 45—say, 60—Vim will resize the window to which you move each time to 60 lines, and it will resize the other window to 30. This is handy for editing two files simultaneously; you automatically increase the allocated window size for maximum context when you switch from window to window and from file to file.

`equalalways`
This tells Vim to always resize windows equally after splitting or closing a window. This is a good option to set in order to ensure equitable allocation of windows as you add and delete them.

`eadirection`
This defines the directional jurisdiction for `equalalways`. The possible values `hor`, `ver`, and `both` tell Vim to make windows of equal size *horizontally*, *vertically*, or *both*, respectively. The resizing applies each time you split or delete a window.

`cmdheight`

This sets the command-line height. As described previously, decreasing a window's height when there is only one window increases the command-line height. You can keep the command line the height you want using this option.

`winminwidth` *and* `winminheight`

These tell Vim the *minimum* width and height to size windows. Vim considers these to be hard values, meaning that windows will never be allowed to get smaller than these values.

Resizing Command Synopsis

Table 10-6 summarizes the ways to resize windows. Options are set with the `:set` command.

Table 10-6. Window resizing commands

Command or option	Type	Description
CTRL-W =	vi Command	Resize all windows equally. The current window honors the settings of the `winheight` and `winwidth` options.
`:resize -n`	ex Command	Decrease the current window size. The default amount is one line.
CTRL-W -	vi Command	Same as `:resize -n`.
`:resize +n`	ex Command	Increase the current window size. The default amount is one line.
CTRL-W +	vi Command	Same as `:resize +n`.
`:resize n`	ex Command	Set the current window height. The default is to maximize window height (unless *n* is specified).
CTRL-W CTRL-_ CTRL-W _	vi Command	Same as `:resize n`.
`z n` ENTER	vi Command	Set the current window height to *n*.
CTRL-W <	vi Command	Decrease the current window width. The default amount is one column.
CTRL-W >	vi Command	Increase the current window width. The default amount is one column.
`:vertical resize n`	ex Command	Set the current window width to *n*. The default is to make the window as wide as possible.
CTRL-W |	vi Command	Same as `:vertical resize n`.
`cmdheight`	Option	Set the command-line height.
`eadirection`	Option	Define whether Vim resizes windows equally vertically, horizontally, or both.
`equalalways`	Option	When the number of windows changes, either by splitting or closing windows, resize them to be the same size.
`winheight`	Option	When entering or creating a window, set its height to at least the specified value.
`winwidth`	Option	When entering or creating a window, set its width to at least the specified value.

Command or option	Type	Description
winminheight	Option	Define the minimum window height, which applies to all windows created.
winminwidth	Option	Define the minimum window width, which applies to all windows created.

Buffers and Their Interaction with Windows

Vim uses *buffers* as containers for work. Understanding buffers completely is an acquired skill; there are many commands for manipulating and navigating them. However, it is worthwhile to familiarize yourself with some of the buffer basics and understand how and why they exist throughout a Vim session.

A good starting point is to open up a few windows editing different files. For example, start Vim by opening *file1*. Then, within the session, issue :split file2 and then :split file3. You should now have three open files in three separate Vim windows.

Now use the commands :ls, :files, or :buffers to list the buffers. You should see three lines, each numbered and including the filenames, along with additional information. These are Vim's buffers for this session. There is one buffer for each file, and each buffer has a unique, nonchanging associated number. In this example, *file1* is in buffer one, *file2* is in buffer two, and *file3* is in buffer three.

Additional information on each buffer can be displayed if you append an exclamation point (!) after any of the commands.

To the right of each buffer number are status flags. These flags describe the buffers, as shown in Table 10-7.

Table 10-7. Status flags describing buffers

Code	Description
u	Unlisted buffer. This buffer is not listed unless you use the ! modifier. To see an example of an unlisted buffer, type :help. Vim splits the current window to include a new window in which the built-in help appears. The plain :ls command will not show the help buffer, but :ls! includes it.
% or #	% is the buffer for the current window. # is the buffer to which you would switch with the :edit # command. These are mutually exclusive.
a or h	a indicates an active buffer. That means the buffer is loaded and visible. h indicates a hidden buffer. The hidden buffer exists but is not visible in any window. These are mutually exclusive.
- or =	- indicates a buffer has the modifiable option turned off. The file is read-only. = is a read-only buffer that cannot be made modifiable (for instance, because you don't have filesystem privileges to write to the file). These are mutually exclusive.
+ or x	+ indicates a modified buffer. x is a buffer with read errors. These are mutually exclusive.

The u flag is an interesting way to know what help file you are viewing in Vim. For example, had you issued `:help split` followed by `:ls!`, you would see that the unlisted buffer refers to the built-in Vim help file, *windows.txt*.

Now that you can list Vim buffers, we can talk about them and their various uses.

Vim's Special Buffers

Vim uses some buffers, called *special buffers*, for its own purposes. For instance, the help buffers described in the previous section are special. Typically, these buffers cannot be edited or modified.

Here are four examples of Vim's special buffers:

directory

Contains directory contents, that is, a list of files for a directory (and some helpful extra command hints). This is a handy tool within Vim that lets you move around in this buffer as you would in a regular text file and select files under the cursor for editing by pressing ENTER.

help

Contains Vim help files, described earlier in the section "Built-In Help" on page 165. `:help` loads these text files into this special buffer.

QuickFix

Contains the list of errors created by your commands (which can be viewed with `:cwindow`) or the location list (which can be viewed with the `:lwindow` command). Do not edit the contents of this buffer! It helps programmers iterate through the edit-compile-debug cycle. See Chapter 11.

scratch

These buffers contain text for general purposes. This text is expendable and can be deleted at any time.

Hidden Buffers

Hidden buffers are Vim buffers that are not currently displayed in any window. This makes it easier to edit multiple files, considering the limited screen real estate for multiple windows, without constantly retrieving and rewriting files. For example, imagine you are editing the *myfile* file but wish to momentarily edit another file, *myOtherfile*. If the `hidden` option is set, you can edit *myOtherfile* through `:edit myOtherfile`, causing Vim to hide the *myfile* buffer and display *myOtherfile* in its place. You can verify this with `:ls` and see both buffers listed, with *myfile* flagged as "hidden."

Buffer Commands

There are almost 50 commands that specifically target buffers. Many are useful but are for the most part outside the scope of this discussion. Vim manages buffers automatically as you open and close multiple files and windows. The suite of buffer commands allows you to do almost anything with buffers. Often they are used within scripts to handle such tasks as unloading, deleting, and modifying buffers.

Two buffer commands are worth knowing for general use because of their power to do lots of work across many files:

`windo` *cmd*
> Short for "window do" (at least we think it's a decent mnemonic), this pseudo-buffer command (actually it's a window command) executes the command *cmd* in each window. It acts as if you go to the top of the screen (CTRL-W T) and cycles through each window to execute the specified command as `:cmd` in that window. It acts only within the current tab and stops at any window where `:cmd` generates an error. The window in which the error occurs becomes the new current window. See the section "Tabbed Editing" on page 233 for a discussion of Vim tabs.
>
> *cmd* is not permitted to change the state of the windows; that is, it cannot delete, add, or change the order of the windows.
>
> > *cmd* can concatenate multiple `ex` commands with the pipe (`|`) symbol. The commands are executed in sequence, with the first command executed sequentially through all windows, then the second command in all windows, and so on.
>
> As an example of `:windo` in action, suppose you are editing a suite of Java files, and for some reason you have a class name that is improperly capitalized. You need to repair this by changing every occurrence of `myPoorlyCapitalizedClass` to `MyPoorlyCapitalizedClass`. With `:windo` you can do that with:
>
> ```
> :windo %s/myPoorlyCapitalizedClass/MyPoorlyCapitalizedClass/g
> ```
>
> Pretty cool!

`bufdo[!]` *cmd*
> This is analogous to `windo` but operates on all of the buffers in your editing session, not just visible buffers in the current tab. `bufdo` stops at the first error encountered, just like `windo`, and leaves the cursor in the buffer where the command fails.
>
> The following example changes all buffers to Unix file format:

```
:bufdo set fileformat=unix
```

Buffer Command Synopsis

Table 10-8 makes no attempt to describe all the commands related to buffers; instead, it summarizes the ones described in this section and some other popular commands.

Table 10-8. Summary of buffer commands

Command	Description

`:ls[!]`
`:files[!]`
`:buffers[!]`
	List buffers and file names. Include unlisted buffers if the `!` modifier is included.

`:ball` `:sball`	Edit all args or buffers. (`sball` opens them in new windows.)
`:unhide` `:sunhide`	Edit all *loaded* buffers. (`sunhide` opens them in new windows.)
`:badd` *file*	Add *file* to list.
`:bunload[!]`	Unload the current buffer from memory. The `!` modifier forces a modified buffer to be unloaded without being saved.
`:bdelete[!]`	Unload the current buffer and delete it from the buffer list. The `!` modifier forces a modified buffer to be unloaded without being saved.
`:buffer [n]` `:sbuffer [n]`	Move to buffer *n*. `sbuffer` opens a new window.
`:bnext [n]` `:sbnext [n]`	Move to the *n*th next buffer. `sbnext` opens a new window.

`:bNext [n]`
`:sbNext [n]`
`:bprevious [n]`
`:sbprevious [n]`
	Move to the *n*th next or previous buffer. `sbNext` and `sbprevious` open a new window.

`:bfirst` `:sbfirst`	Move to the first buffer. `sbfirst` opens a new window.
`:blast` `:sblast`	Move to the last buffer. `sblast` opens a new window.
`:bmod [n]` `:sbmod [n]`	Move to the *n*th modified buffer. `sbmod` opens a new window.

Playing Tag with Windows

Vim extends the `vi` tag functionality into windows by offering the same tag traversal mechanisms through multiple windows. (See the section "Using Tags" on page 147 for a discussion of `vi` tags.) Following a tag can also open a file in the associated place in a new window.

The tag windowing commands split the current window and follow a tag either to a file matching the tag or to the file matching the filename under the cursor:

`:stag[!]` *tag*

    This splits the window to display the location for the tag found. The new file containing the matched tag becomes the current window, and the cursor is placed over the matched tag. If no tag is found, the command fails, and no new window is created.

 As you become more familiar with Vim's help system, you can use `:stag` to split your way through the help system rather than jumping from file to file in the same window.

`CTRL-W` `]` *or* `CTRL-W` `^`

    These split the window and open a window above the current window. The new window becomes the current window, and the cursor is placed on the matching tag. If there is no match for the tag, the command fails.

`CTRL-W` `G`

    This splits the window and creates a new window above the current window. In the new window, Vim performs the command `:tselect` *tag*, where *tag* was the tag identifier under the cursor. The cursor is placed in the new window, and that new window becomes the current window. If no matching tag exists, the command fails.

`CTRL-W` `G` `CTRL-]`

    This works exactly like `CTRL-W` `G`, except that instead of performing `:tselect`, it performs `:tjump`.

`CTRL-W` `F` *or* `CTRL-W` `CTRL-F`

    These split the window and edit the filename underneath the cursor. Vim looks sequentially through the files set in the option variable `path` to find the file. If the file doesn't exist in any of the `path` directories, the command fails and does not create a new window.

`CTRL-W` `SHIFT-F`

    This splits the window and edits the filename under the cursor. The cursor is placed in the new window editing that file and positioned at the line number matching the number following the filename in the first window.

`CTRL-W` `G` `F`

    This opens the file under the cursor in a new tab. If the file doesn't exist, the new tab is not created.

CTRL-W   G   SHIFT-F

> This opens the file under the cursor in a new tab and positions the cursor on the line specified by the number following the filename in the first window. If the file doesn't exist, the new tab is not created.

Tabbed Editing

Did you know that in addition to editing in multiple windows, you can create multiple *tabs*? Vim lets you create new tabs, each of which behaves independently. In each tab you can split the screen, edit multiple files—virtually anything you would normally do in a single window; but now all of your work is easily managed in one window with tabs.

Many Chrome and Firefox users are very familiar with and dependent on tabbed browsing and will recognize the value that this feature brings to power editing.[4] For the uninitiated, it's worth trying.

You can use tabs in both regular Vim and `gvim`, but `gvim` is much nicer and easier. Some of the more important ways to create and manage tabs include:

`:tabnew` *filename* or `:tabedit` *filename*
> Open a new tab and edit a file (optional). If no file is specified, Vim opens a new tab with an empty buffer.

`:tabclose`
> Close the current tab.

`:tabonly`
> Close all other tabs. If other tabs have modified files, they are not removed unless the `autowrite` option is set, in which case all modified files are written before the other tabs are closed.

In `gvim` you can activate any tab simply by clicking the tab at the top of the screen. You can also activate tabs in character-based terminals with the mouse if the mouse is configured (see the `mouse` option). Also, it's easy to move right or left from tab to tab with CTRL PAGEDOWN (move one tab to the right) and CTRL PAGEUP (move one tab to the left). If you are in the leftmost or rightmost tabs and you try to move left or right, respectively, Vim moves to the far right or far left tab.

`gvim` offers right-click pop-up menus for the tab, from which you can open a new tab (with or without a new file to edit) and close a tab.

---

4 Wow, when the seventh edition of this book was released, Chrome did not even exist! And today all browsers have tabs, and all users should be familiar with them.

Figure 10-4 is an example of a set of tabs (notice the tab pop-up menu). Figure 10-5 is the same example in a terminal emulator.

Figure 10-4. Example of gvim tabs and tabbed editing

Figure 10-5. Same tabbed example in a terminal emulator (Linux Vim)

The Vim command-line option -p opens multiple files, causing each file to occupy a separate tab. Our examples in Figures 10-4 and 10-5 were invoked this way:

```
gvim -p ~/.vimrc ~/.gvimrc /etc/hosts /etc/resolv.conf/etc/sysctl.conf
vim -p ~/.vimrc ~/.gvimrc /etc/hosts /etc/resolv.conf/etc/sysctl.conf
```

Closing and Quitting Windows

There are four different ways to close a window that are specific to window editing—*quit*, *close*, *hide*, and *close all others*:

CTRL-W Q *or* CTRL-W CTRL-Q

These are really just window versions of the :quit command. In their simplest form (i.e., in a single session with only one window), they behave exactly like

vi's `:quit` command. If the `hidden` option is set and the current window is the last window on the screen referencing that file, the window is closed but the file buffer is retained and hidden (it can be retrieved). In other words, Vim is still storing the file, and you can return to editing it later. If `hidden` is *not* set, the window is the last one referencing that file, and there are unsaved changes, the command fails in order to avoid losing your changes. But if some other window displays the file, the current window closes.

CTRL-W C *or* `:close[!]`

These close the current window. If the `hidden` option is set and this is the last window referencing this file, Vim closes the window and the buffer is hidden. If this window is on a tab and is the last window for that tab, the window *and* the tab are closed. As long as you don't use the `!` modifier, this command will not abandon any file with unsaved changes. The `!` modifier tells Vim to close the current window unconditionally.

> Note that this command does not use CTRL-W CTRL-C, because Vim uses CTRL-C to *cancel* commands. Therefore, if you try to use CTRL-W CTRL-C, the CTRL-C simply cancels the command.
>
> Similarly, while the CTRL-W commands are used in combination with CTRL-S and CTRL-Q, *some* users may find their terminal emulators frozen because some terminal emulators interpret CTRL-S and CTRL-Q as control characters to stop and start displaying information to the screen. If you find your screen freezing mysteriously when using these combinations, try the other listed combinations instead.

CTRL-W O , CTRL-W CTRL-O , *and* `:only[!]`

These close all windows except the current window. If the `hidden` option is set, all closed windows hide their buffers. If it's not set, any window referencing a file with unsaved changes remains on the screen, unless you include the `!` modifier, in which case all windows are closed and the files are abandoned. The behavior of this command can be affected by the `autowrite` option: if it's set, all windows are closed, but windows containing unsaved changes are written to their files on disk before being exited.

`:hide` *[cmd]*

This quits the current window and hides the buffer if no other window references it. If the optional *cmd* is supplied, the buffer is hidden, and the command is executed.

Table 10-9 provides a summary of these commands.

Table 10-9. Commands for closing and quitting windows

Command	Keystrokes	Description
`:quit[!]`	CTRL-W Q CTRL-W CTRL-Q	Quit the current window.
`:close[!]`	CTRL-W C	Close the current window.
`:only[!]`	CTRL-W O CTRL-W CTRL-O	Make the current window the only window.
`:hide[cmd]`		Close the current window and hide the buffer. Execute *cmd* if present.

Summary

As you now appreciate, Vim ramps up the editing horsepower with its many windowing features. Vim lets you create and delete windows easily and on the fly. Additionally, Vim provides the under-the-hood power of the raw buffer commands, buffers being the underlying file management infrastructure with which Vim manages window editing. This is once again a perfect example of how Vim brings multiwindow editing to beginners while simultaneously giving power users the tools they need to tune their windowing experience.

Vim Enhancements for Programmers

Text editing is only one of Vim's strong suits. Good programmers demand powerful tools to ensure efficient and proficient work. A good editor is only a start and by itself isn't enough. Many modern programming environments attempt to provide comprehensive solutions, when all that is really necessary is a powerful and efficient editor with some extra smarts.

Programming tools offer extra features ranging from editors with syntax coloring, auto indentation and formatting, keyword completion, and so on to full-blown integrated development environments (IDEs) with sophisticated integration that build up complete development ecosystems. These IDEs can be expensive (e.g., Visual Studio[1]) or free (Eclipse), and even though computer resource demands aren't as dominant, often something lightweight is sufficient. Vim fulfills the *lightweight* space by providing some IDE-ish features and, with community-provided plug-ins, approaches IDE functions. (For a deeper dive into developing with Vim IDE plug-ins, see Chapter 15, "Vim as IDE: Some Assembly Required".)

Programmers' tasks vary, and so do their technology requirements. Small development tasks are easily completed with simple editors that offer little more than text editing capabilities. Large multicomponent, multiplatform, and multistaff efforts *almost* demand the heavy lifting IDEs provide. But from anecdotal experience, many veteran programmers feel that IDEs offer little more than extra complexity without higher probability of success.

Vim strikes a nice compromise between simple editors and monolithic IDEs. It has features that until recently were available only in expensive IDEs. It lets you do quick and simple programming tasks without the overhead and learning curve of an IDE.

---

1 Not to be confused with Microsoft's VS Code, which is free and excellent.

The many options, features, commands, and functions especially suited to making the programmer's life easier range from folding lines of code into one line to syntax coloring and automatic formatting. Vim affords programmers many tools that can be fully appreciated only by using them. At the high end, it offers a sort of mini-IDE called QuickFix, but it also has convenience features specific to various programming tasks. We present the following topics in this chapter:

- Folding
- Auto and smart indenting
- Keyword and dictionary word completion
- Tags and extended tags
- Syntax highlighting and highlight authoring (rolling your own)
- QuickFix, Vim's mini-IDE

Folding and Outlining (Outline Mode)

Folding lets you define what parts of the file you see. For instance, in a block of code you can hide anything within curly braces, or hide all comments. Folding is a two-stage process. First, using any of several fold methods (which we describe soon), you define what constitutes a block of text to fold. Then, when you use a fold command, Vim hides the designated text and leaves in its place a one-line placeholder. Figure 11-1 shows what folds look like in Vim. You can manage the lines hidden by the fold with the fold placeholder.

```
   4
   5  int fcn (int v1, int v2)
   6    {
   7
   8      printf ("02 some line\n");
   9      printf ("03 some line\n");
  10 +--   2 lines: printf ("04 some line\n");-------------------
  12      printf ("06 some line\n");
  13
  14      if (thiscode == anysense)
  15 +--   8 lines: { --------------------------------------------
  23
  24      printf ("06 some line\n");
  25      printf ("06 some line\n");
  26 +--   4 lines: printf ("06 some line\n");-------------------
  30
  31    }
```

Figure 11-1. Example of Vim folds (MacVim, color scheme: zellner)

In the example, line 11 is hidden by a two-line fold starting with line 10. An eight-line fold starting at line 15 hides lines 15 through 22. And a four-line fold starting at line 26 hides lines 26 through 29.

There are virtually no limits on how many folds you can create. You can even create nested folds (folds within folds).

Several options control how Vim creates and displays folds. Also, if you've taken the time to create many folds, Vim provides the convenience commands `:mkview` and `:loadview` to preserve folds between sessions so that you don't have to create them again.

Folds require some effort to learn but, when mastered, add a powerful way to control what to display and when. Do not underestimate the power this brings. Correct and maintainable programs require robust design at several levels, so good programming often requires looking at the forest rather than the trees[2]—in other words, ignoring details of implementation in order to see the overall structure of a file.

For power users, Vim offers six different ways to define, create, and manipulate folds. This flexibility lets you create and manage folds in different contexts. Ultimately, once created, folds open and close and behave similarly for the whole suite of fold commands.

The six methods of creating folds are:

diff
    The differences between two files define folds.

expr
    Regular expressions define folds.

indent
    Folds and fold levels correspond to the indentation of text and the value of the option `shiftwidth`.

manual
    Folds and fold levels result from user Vim commands (for example, folding a paragraph).

marker
    Predefined (but also user-definable) markers in the file specify fold boundaries.

---

2 Perhaps this might actually be "requires looking at the trees rather than the forest." Maybe it's both. Folds give you that!

syntax
    Folds correspond to the semantics of a file's language (e.g., a C program's function blocks could fold).

You use these terms as the value of the foldmethod option. The manipulation of folds (opening and closing, deleting, etc.) is the same for all methods. We'll examine manual folds and discuss Vim fold commands in detail. We address some details for the other methods, but they are complex, specialized, and beyond the scope of this introduction. We hope our coverage will prompt you to explore the richness of these other methods.

So let's take a brief look at the important fold commands and go through a short example of what folds look like.

The Fold Commands

Fold commands all begin with z. As a mnemonic to remember this, think of the side view of a folded piece of paper (when folded correctly) and how it looks like the letter "z."

There are about 20 z fold commands. With these commands you create and delete folds, open and close folds (hide and expose text belonging to folds), and toggle the expose/hide state of the folds. Here are short descriptions:[3]

zA
    Toggle the state of folds, recursively.

zC
    Close folds, recursively.

zD
    Delete folds, recursively.

zE
    Delete *all* folds.

zf
    Create a fold from the current line to the one where the following motion command takes the cursor.

[*count*] zF
    Create a fold covering *count* lines, starting with the current line.

---

3 Please note and take care not to confuse "deleting folds" with Vim's delete command. Deleting a fold removes the visual semantic of hidden lines. Deleting the content of a fold does just that!

zM

    Set option `foldlevel` to zero.

zN, zn

    Set (`zN`) or reset (`zn`) the `foldenable` option.

zO

    Open folds, recursively.

za

    Toggle the state of one fold.

zc

    Close one fold.

zd

    Delete one fold.

zi

    Toggle the value of the `foldenable` option.

zj, zk

    Move the cursor to the start (`zj`) of the next fold or to the end (`zk`) of the previous fold. Note the mnemonic of the `j` and `k` motion commands and how they are analogous to motions within the context of folds.

zm, zr

    Decrement (`zm`) or increment (`zr`) the value of the `foldlevel` option by one.

zo

    Open one fold.

Don't confuse *deleting a fold* with *deleting text*. Use the "delete fold" (`zd`) command to remove, or undefine, a fold. A deleted fold has no effect on the text contained in that fold. You may notice we mention this more than once. It's important to know. One of us has lost work thinking he merely deleted folds when, in fact, he deleted content. Of course this is something that is always discovered inconveniently late.

`zA`, `zC`, `zD`, and `zO` are called recursive because they operate on all folds nested within the one where you issue the commands.

Manual Folding

If you know Vim motion commands, you already know half of what you must learn to be proficient with manual fold commands.

For example, to hide three lines in a fold, enter either of the following:

```
3zF
2zfj
```

`3zF` executes the `zF` folding command over three lines, starting with the current one. `2zfj` executes the `zf` folding command from the current line to the line where `j` moves the cursor (two lines down).

Let's try a more sophisticated command of use to C programmers. To fold a block of C code, position the cursor over the beginning or ending brace (`{` or `}`) of a block of code and type `zf%`. (Remember that `%` moves to the matching brace.)

Create a fold from the cursor to the beginning of the file by typing `zfgg`. (`gg` goes to the beginning of the file.)

It is easier to understand folds by seeing an example. We'll take a simple file, create and manipulate folds, and watch the behavior. We'll also see some of the enhanced visual folding cues that Vim provides.

First consider the example file in Figure 11-2, which contains some (meaningless) lines of C code. Initially, there are no folds.

Figure 11-2. Sample file with no folds (MacVim, color scheme: zellner)

There are a few things to note in this picture. First, Vim displays line numbers on the left side of the screen. We recommend that you always turn them on (using the number option) for added visual information about file location, and in this context the information becomes more valuable when you fold lines out of view. Vim tells you how many lines are not displayed, and the line numbers confirm and reinforce that information.

Also notice the gray columns to the left of the line numbers. These columns are reserved for more folding visual cues. As we create and use folds, we will see the visual cues that Vim inserts into these columns.

In Figure 11-2, notice that the cursor is on line 18. Let's fold that line and the two following lines into one fold. We type zf2j. Figure 11-3 shows the result.


```
   1
   2
   3
   4
   5 int fcn (int v1, int v2)
   6   {
   7
   8    printf ("02 some line\n");
   9    printf ("03 some line\n");
  10    printf ("04 some line\n");
  11    printf ("05 some line\n");
  12    printf ("06 some line\n");
  13
  14    if (thiscode == anysense)
  15       {
  16
  17       printf ("07 some other line\n");
+ 18 +--   3 lines: printf ("08 some other line\n");-------------------
  21
  22       }
  23
  24    printf ("07 some line\n");
  25    printf ("08 some line\n");
  26    printf ("09 some line\n");
  27    printf ("10 some line\n");
  28    printf ("11 some line\n");
  29    printf ("12 some line\n");
  30
  31    }
```

Figure 11-3. Three lines folded at line 18 (MacVim, color scheme: zellner)

Notice how Vim creates an easily identified marker with the +-- as a prefix, and how it displays text from the first folded line in the *fold* placeholder. Now notice where Vim inserted the + at the far left side of the screen. This is another visual cue.

In the same file, we'll next fold the block of code between and including the braces after the if statement. Position the cursor on either one of the braces and type zf%.[4] The file now appears as in Figure 11-4.

---

4 Folding is general; you can use the concept of text objects introduced in Part I. Thus, you can fold anywhere within an *object*—in this case, anywhere within the braces. The vi command would be za{.

```
                     10   printf ("04 some line\n");
                     11   printf ("05 some line\n");
                     12   printf ("06 some line\n");
                     13
                     14   if (thiscode == anysense)
   +                 15 +--   8 lines: { ---------------------------
                     23
                     24   printf ("07 some line\n");
                     25   printf ("08 some line\n");
                     26   printf ("09 some line\n");
                     27   printf ("10 some line\n");
                     28   printf ("11 some line\n");
                     29   printf ("12 some line\n");
                     30
                     31   }
```

Figure 11-4. Block of code folded following an `if` *statement (MacVim, color scheme: zellner)*

Now there are eight lines of code folded, three of which are contained in a previously created fold. This is called a *nested* fold. Note that there is no indication of the nested fold.

Our next experiment is to position the cursor on line 25 and fold all lines up to and including the function declaration for `fcn`. This time we use the Vim *search* motion. We initiate the fold command with `zf`, search backward to the beginning of the `fcn` function using `?int fcn` (the backward search command in Vim), and press the ENTER key. The screen now looks like Figure 11-5.

```
                      3
                      4
   +                  5 +-- 21 lines: int fcn (int v1, int v2)---
                     26   printf ("09 some line\n");
                     27   printf ("10 some line\n");
                     28   printf ("11 some line\n");
                     29   printf ("12 some line\n");
```

Figure 11-5. Folding to the beginning of a function (MacVim, color scheme: zellner)

> If you count lines and create a fold that spans another fold (for example, `3zf`), all lines contained in the spanned fold count as one line. For example, if the cursor is on line 30, and lines 31–35 are hidden in a fold on the next screen line so that the next line on the screen displays line 36, `3zf` creates a new fold containing three lines as they appear on the screen: the text line 30, the five lines contained in the fold holding lines 31–35, and the text line 36 displayed in the next line on the screen. Confusing? A little. You might say that the `zf` command counts lines in accordance with the rule "What you see is what you fold."

Let's try some other features. First, open all the folds with the command `zO` (that's `z` followed by the letter O, not `z` followed by a zero). Now we start seeing some visual

cues in the left margin about the folds we made, as shown in Figure 11-6. Each of the columns in this margin is called a fold column.

```
     4
-    5  int fcn (int v1, int v2)
|    6  {
|    7
|    8    printf ("02 some line\n");
|    9    printf ("03 some line\n");
|   10    printf ("04 some line\n");
|   11    printf ("05 some line\n");
|   12    printf ("06 some line\n");
|   13
|   14    if (thiscode == anysense)
|-  15      {
||  16
||  17      printf ("07 some other line\n");
||- 18      printf ("08 some other line\n");
|||  19     printf ("09 some other line\n");
|||  20     printf ("10 some other line\n");
||  21
||  22      }
|   23
|   24    printf ("07 some line\n");
|   25    printf ("08 some line\n");
    26    printf ("09 some line\n");
    27    printf ("10 some line\n");
    28    printf ("11 some line\n");
    29    printf ("12 some line\n");
    30
    31  }
```

Figure 11-6. All folds opened (MacVim, color scheme: zellner)

In this figure, the first line of each fold is marked with a minus sign (-), and all the other lines of the fold are marked by a vertical bar or pipe character (|). The largest (outermost) fold is in the leftmost column, and the innermost fold is in the rightmost column. As you see in our picture, lines 5–25 represent the lowest fold level (in this case, one), lines 15–22 represent the next fold level (two), and lines 18–20 represent the highest fold level.

> By default, this helpful visual metaphor is turned off (we don't know why—perhaps because it uses up screen space). Turn it on and define its width with the following command:
>
>   :set foldcolumn=n
>
> where *n* is the number of columns to use (maximum is twelve, default is zero). In the figure, we use foldcolumn=5. For those paying close attention, yes, the earlier figures had foldcolumn set to three. We changed the value for a better visual presentation.

Now create more folds to observe their effects.

First, refold the deepest fold, which covers lines 18–20, by positioning the cursor on any line within the range of that fold and typing zc (close fold). Figure 11-7 shows the result.

```
     .
 -   5  int fcn (int v1, int v2)
 |   6  {
 |   7
 |   8    printf ("02 some line\n");
 |   9    printf ("03 some line\n");
 |  10    printf ("04 some line\n");
 |  11    printf ("05 some line\n");
 |  12    printf ("06 some line\n");
 |  13
 |  14    if (thiscode == anysense)
 |- 15      {
 ||  16
 ||  17      printf ("07 some other line\n");
 ||+ 18 +----  3 lines: printf ("08 some other line\n");----------
 ||  21
 ||  22      }
 |   23
 |  24    printf ("07 some line\n");
 |  25    printf ("08 some line\n");
 |  26    printf ("09 some line\n");
 |  27    printf ("10 some line\n");
 |  28    printf ("11 some line\n");
 |  29    printf ("12 some line\n");
 |  30
 |  31  }
```

Figure 11-7. After refolding lines 18–20 (MacVim, color scheme: zellner)

See the change in the gray margin? Vim maintains the visual cues, making visualization and management of your folds easy.

Now let's see what a typical "one line" command does to a fold. Position the cursor on the folded line (18). Type ~~ (toggle case for all characters in the current line). Note that this is the usage for changing case inline when the Vim option `tildeop` is set; otherwise the command would be g~~. Remember that in Vim, ~ is an object operator (unless the `compatible` option is set) and therefore should toggle the case of all the characters in the line. Next, open the fold by typing zo (open fold). The code now looks like Figure 11-8.

```
 ||  14    if (thiscode == anysense)
 |- 15      {
 ||  16
 ||  17      printf ("07 some other line\n");
 ||- 18 █    PRINTF ("08 SOME OTHER LINE\N");
 |||  19      PRINTF ("09 SOME OTHER LINE\N");
 |||  20      PRINTF ("10 SOME OTHER LINE\N");
 ||  21
 ||  22      }
```

Figure 11-8. Case change applied to a fold (MacVim, color scheme: zellner)

This is a powerful feature. Line commands or operators act on the entire text represented by a fold line! Admittedly this may seem like a contrived example, but it nicely illustrates the potential of this technique.

> As we just saw, any action on a fold affects the whole fold. For instance, in Figure 11-7, if you position the cursor over line 18—a fold hiding lines 18 through 20—and type dd (delete line), all three lines are deleted and the fold is removed.
>
> It's also important to note that Vim manages all edit actions as if there were no folds, so any undos will undo an entire edit's action. So if we typed u (undo) after the previous change, all three lines that had been deleted would be restored. The undo feature is separate from the "one line" actions discussed in this section, although sometimes they seem to act similarly.

Now is a good time to familiarize yourself with the visual cues in the fold column margin. They make it easy to see what fold you are about to act on. For example, the zc (close fold) command closes the innermost fold containing the line the cursor is on. You can see how large this fold is through the vertical bars in the fold columns. Once mastered, actions such as opening, closing, and deleting folds become second nature.

Outlining

Consider the following simple (and contrived) file using tabs for indentation:

```
1. This is Headline ONE with NO indentation and NO fold level.
    1.1 This is sub-headline ONE under headline ONE
        This is a paragraph under the headline.  Its fold
        level is 2.
    1.2 This is sub-headline TWO under headline ONE.
2. This is Headline TWO.  No indentation, so no folds!
    2.1 This is sub-headline ONE under headline TWO.
        Like the indented paragraph above, this has fold level 2.
            - Here is a bullet at fold level 3.
                A paragraph at fold level 4.
            - Here is the next bullet, again back at fold level 3.
        And, another set of bullets:
            - Bullet one.
            - Bullet two.
    2.2 This is sub-heading TWO under Headline TWO.
3. This is Headline THREE.
```

You can use Vim folds to look at your file as a pseudo-outline. Define your folding method as indent:

```
:set foldmethod=indent
```

In our file we define the shiftwidth (the indentation level for tabs) to be 4. Now we can open and close folds based on indentation of lines. For each shiftwidth (a

multiple of four columns in this case) to a line that is indented, its fold level increases by one. For example, the subheadlines in our file are indented one shiftwidth, or four columns, and hence have a fold level of one. Lines indented eight columns (two shiftwidths) have a fold level of two, and so on.

You can control the level of folds you see with the `foldlevel` command. It takes an integer as its argument and displays only lines whose fold levels are *less than or equal to* the argument. In our file we can ask to view only the highest-level headings with:

```
:set foldlevel=0
```

and our screen now looks like Figure 11-9.

Figure 11-9. Results of `:set foldlevel=0` (Linux `gvim`, color scheme: zellner)

Display everything up to and including the bullets by setting `foldlevel` to two. Everything with a fold level *greater than or equal to* two is then displayed, as in Figure 11-10.

Using this technique to inspect your file, you can quickly expand and collapse the level of detail you see with Vim's fold increment (`zr`) and decrement (`zm`) commands.

Figure 11-10. Results of `:set foldlevel=2` (Linux `gvim`, color scheme: zellner)

A Few Words About the Other Fold Methods

We can't cover all of the other fold methods, but to whet your appetite, we'll take a quick peek at the syntax folding method.

We use the same C file as before, but this time we let Vim decide what to fold based on C syntax. The rules governing folding within C are complex, but this simple snippet of code suffices to demonstrate Vim's automatic capabilities.

First, make sure to get rid of all folds by typing zE (eliminate all folds). The screen now displays all code with no visual markers in the fold column.

Make sure folding is turned on with the command:

```
:set foldenable
```

You didn't need to do this before for manual folding, because by default foldenable is enabled and foldmethod is set to manual. Now enter these two Vim ex commands:

```
:syntax on
:set foldmethod=syntax
```

The folds appear as in Figure 11-11.

Figure 11-11. After the command `:set foldmethod=syntax` (Linux gvim, color scheme: zellner)

Vim folded all bracketed blocks of code, because those are logical semantic blocks in C. If you type zo on line 6 of this example, Vim expands the fold and reveals the inner fold.

Each fold method uses different rules to define folds. We encourage you to roll up (fold up?) your sleeves and read more on these powerful methods in the Vim documentation.

The Vim diff mode (also invoked through the vimdiff command) is a powerful combination of folding, windowing, and syntax highlighting, a feature we discuss later. As illustrated in Figure 11-12, the mode shows the differences between files, usually between two versions of the same file.

Figure 11-12. Vim diff feature and its use of folds (Linux `gvim`*, color scheme: zellner)*

Auto and Smart Indenting

Vim offers four increasingly complex and powerful methods to automatically indent text. In its simplest form, Vim behaves almost identically to `vi`'s `autoindent` option, and indeed it uses the same name to describe the behavior. (See the section "Indentation Control" on page 143 for information on how `vi` does automatic indentation.)

You can choose the indentation method simply by specifying it in a `:set` command, such as:

```
:set cindent
```

Vim offers the following methods, listed in order of increasing sophistication:

`autoindent`
Auto indentation closely mimics `vi`'s `autoindent`. It differs subtly as to where the cursor is placed after indentation is deleted.

`smartindent`
This is slightly more powerful than `autoindent`, but it recognizes some basic C syntax primitives for defining indentation levels.

`cindent`
As its name implies, `cindent` embodies a much richer awareness of C syntax and introduces sophisticated customization beyond simple indentation levels. For example, `cindent` can be configured to match your (or your boss's) favorite coding style rules, including but not limited to how braces ({}) indent, where braces are placed, whether either or both braces are indented, and even how indentation matches included text.

`indentexpr`
This lets you define your own expression, which Vim evaluates in the context of each new line you begin. With this feature, you write your own rules. We refer you to Chapter 12, "Vim Scripts", and to the Vim documentation for details. If the other three options don't give you enough flexibility for automatic indentation, `indentexpr` certainly will.

Vim autoindent Extensions to vi's autoindent

autoindent for Vim behaves almost like vi's and can be made identical by setting the compatible option. One nice extension to vi's autoindent is Vim's ability to recognize a file's "type" and insert appropriate comment characters when comment lines in a file wrap to a new line. This feature works cooperatively with either the wrapmargin option (text wraps within wrapmargin columns of the right margin) or the textwidth option (text wraps when characters in a line exceed textwidth characters). Figure 11-13 shows the results of identical inputs, one using Vim's autoindent and the other using vi.

```
 1 #! /bin/sh
 2
 3 if [ $xyz -eq 0 ]
 4 then
 5     # this block of comments I typed
 6     # with option textwidth set to 40,
 7     # autoindent on, and
 8     # (automatically), syntax=sh.
 9     # Notice how each line has the '#'
10     # with a separating space, all
11     # courtesy of vim's autoindent...
12     # now I will type the same text but
13     # instead with the option
14     # "compatible" (with vi) set...
15
16     # this block of comments I typed
17     with option textwidth set to 40,
18     autoindent on, and (automatically),
19     syntax=sh.  Notice how each line has
20     the '#' with a separating space, all
21     courtesy of vim's autoindent... now
22     I will type the same text but
23     instead with the option "compatible"
24     (with vi) set...
```

Figure 11-13. Difference between Vim and vi autoindent (Linux gvim, color scheme: zellner)

Notice that in the second block of text (line 16 and beyond) there is no leading comment character. Also, with the compatible option set (to mimic vi's behavior), the textwidth option isn't recognized, and now the text wraps only because option wrapmargin has a value.

smartindent

smartindent extends autoindent, slightly. It's useful, but if you are writing code in a C-like programming language with a fairly complex syntax, you are better served by using cindent instead.

smartindent automatically inserts indents when:

- A new line follows a line with a left brace ({).

- A new line begins with a keyword that's contained in the option `cinwords`.

- A new line is created preceding a line starting with a right brace (}), *if* the cursor is positioned on the line containing the brace and the user creates a new line using the O (open line above) command.

- A new line is a closing, or right, brace (}).

> Normally, you should turn on `autoindent` when using `smartindent`:
>
>   :**set** autoindent

cindent

Regular Vim users who program in C-like languages will want to use either `cindent` or `indentexpr` for coding. Although `indentexpr` is more powerful, flexible, and customizable, `cindent` is more practical for most programming tasks. It has plenty of settings for most programmers' needs (and corporate standards). Try it for a while with its default settings, and then customize it if your standards differ.

> If the `indentexpr` option is nonempty, it overrides `cindent`'s actions.

Three options define `cindent`'s behavior:

`cinkeys`
  Defines keyboard keys that signal Vim to reevaluate indentation.

`cinoptions`
  Defines the indentation style.

`cinwords`
  Defines keywords that signal when Vim should add an extra indent in subsequent lines.

`cindent` uses the string defined by `cinkeys` as its ruleset to define how to indent. We'll examine the default value of `cinkeys` and then look at other settings you can define and how they work.

The cinkeys option

`cinkeys` is a comma-separated list of values:

```
0{,0},0),:,0#,!^F,o,O,e
```

Here are the values, broken into their separate contexts, with brief descriptions for each behavior:

0{

0 (zero) sets a *beginning of line* context for the following character, {. That is, if you type the character { as the first character of a line, Vim will reevaluate the indentation for that line.

Do not confuse the zero in this option with the behavior "use zero indentation here," a common practice in C indentation. The zero here means "if the character is typed at the beginning of the line," not "force the character to appear at the beginning of the line."

The default indentation for { is zero: no added indentation beyond the current level. The following example shows typical results:

```
main ()
{
    if ( argv[0] == (char *)NULL )
    { ...
```

0}, 0)

As in the previous description, these two settings define *beginning of line* context. Thus, if you type either } or) at the beginning of a line, Vim reevaluates indentation.

The default indentation for } matches the indentation defined for its matching open brace. The default indentation for) is one `shiftwidth`.

:

This is the C label or `case`-statement context. If a : (colon) is typed at the end of a label or `case` statement, Vim reevaluates indentation.

The default indentation for : is column one, the first column in a line. Do not confuse this with zero indentation, which leaves the new line at the same indentation level as the previous one. When the indentation is one, the first character of a new line is shifted left *all the way* to the first column.

0#

Again, this is a *beginning of line* context. When # is the first character typed in a line, Vim reevaluates indentation.

Default indentation, as in the previous definition, shifts the entire line to the first column. This is consistent with the practice of beginning macro definitions (#define …) in column one.

!^F

The special character ! defines any following character as a trigger to reevaluate the indentation in the current line. In this case, the triggering character is ^F, which stands for CTRL-F, so the default behavior is for Vim to reevaluate a line's indentation any time you type CTRL-F.

o

This context defines any new line you create, whether by pressing the ENTER key in *insert* mode or by using the o (open new line) command.

O

This context covers the creation of a new line *above* the current line using the O (open new line above) command.

e

This is the *else* context. If you begin a line with the word else, Vim reevaluates indentation. Vim does not recognize this context until the final "e" of *else* is typed.

cinkeys syntax rules

Each cinkeys definition consists of an optional prefix (one of !, *, or 0) and the key for which indentation is reevaluated. The prefixes have the following meanings:

!

Indicates a key (default CTRL-F) that causes Vim to reevaluate indentation on the current line. You can add an additional key definition as a command (by using the += syntax) without overriding the preexisting command. In other words, you can provide multiple keys to trigger line indentation. Any key you add to the ! definition still performs its old function as well.

*

Tells Vim to reevaluate the current line indentation before inserting the key.

0

Sets a *beginning of line* context. The key you specify after the 0 triggers a reevaluation of indentation only when typed as the first character of a line.

Be aware of the distinction in vi and Vim between "first character in a line" and "first column in a line." You already know that typing ^ moves to the first character of a line, not necessarily to the first column (flush left); the same is true of inserting with I. In the same way, the 0 prefix applies to entering a character as the first character in a line, regardless of whether it is flush left.

cinkeys has special key names and provides ways to override any reserved characters, such as those used as prefix characters. Here are the special key options:

<>
Use this form to define keys literally. For special nonprinting keys, use the spelled-out versions. For example, you can define the literal character ":" with <:>. Another example for a nontyping key is to define the "up arrow" as <Up>.

^
Use the caret (^) to signify a control character. For example, ^F defines the key CTRL-F.

o, 0, e, :
We saw these special keys in the default value for cinkeys.

= *word*, =~ *word*
Use these to define a word that should receive special behavior. Once the string *word* is matched, if it is the first text on a new line, Vim reevaluates indentation.

The form =~*word* is the same as =*word* except that it ignores case.

The term *word* is an unfortunate misnomer. More properly, it represents *beginning of word*, because the trigger occurs as soon as the string matches, but it does not require that the matched end of string also be the end of a word. Vim's documentation gives the example of end matching both end and endif.

The cinwords option

cinwords defines keywords that, when typed, trigger extra indentation on the following line. The option's default value is:

```
if,else,while,do,for,switch
```

This covers the standard keywords in the C programming language.

These keywords are case sensitive. In checking for them, Vim even ignores the setting of the `ignorecase` option. If you need variations for different cases of keywords, you must specify all combinations in the `cinwords` string.

The cinoptions option

`cinoptions` controls how Vim reindents lines of text in their C context. It includes settings to control a number of code formatting standards, such as:

- How far to indent a code block enclosed by braces
- Whether to insert a newline in front of a brace that follows a condition statement
- How to align blocks of code relative to their enclosing braces

`cinoptions` defines 28 settings with its default value:

```
s,e0,n0,f0,{0,}0,^0,:s,=s,l0,b0,gs,hs,ps,ts,is,+s,c3,C0,/0,(2s,us,U0,w0,W0,
    m0,j0,)20,*30
```

The very length of the option gives you a sense of how many ways Vim lets you customize indentation. Most customizations with `cinoptions` define slight differences in context blocks. Some customizations define how far to scan (how many lines forward and backward in the file to go) in order to establish the right context and properly evaluate indentation.

Settings that alter the amount of indentation for various contexts can increase or decrease levels of indentation. Also, you can redefine the number of columns to use for indentation. For example, setting `cinoptions=f5` causes an opening brace ({) to be indented five columns, so long as it is not inside any *other* braces.

Another way to define increments of indentation is by some multiplier (which doesn't have to be an integer) of `shiftwidth`. If, in the previous example, you append `w` to the definition (i.e., `cinoptions=f5w`), the opening brace shifts five shiftwidths.

Insert a minus sign (-) before any numeric value to alter the indentation level to the left (a negative indentation).

This option and its string value should be modified with great care. Remember that when you use = syntax, you redefine an option completely. Because `cinoptions` carries so many possible settings, use very fine-grained commands to make changes: += to add a setting, -= to remove an existing setting, and -= followed by += to change an existing setting.

The following is a brief list of the options you are most likely to change. It is a small subset of the settings in `cinoptions`, and you may find the other (or even *all*) settings useful to customize:

>n (default is s)

Any line for which indentation is indicated should be indented *n* places. The default for this is s, meaning that the default indentation for a line is one shiftwidth.

f n, { n

The f defines how far to indent an opening unnested brace ({). The default value is zero, thus aligning braces with their logical counterpart. For example, a brace following a line with a `while` statement is placed under the *w* of the `while`.

The { behaves the same way as the f but applies to *nested* opening braces. Again, this one defaults to an indent level of zero.

Figures 11-14 and 11-15 show two examples of identical text entry in Vim—the first example with `cinoptions=s,f0,{0`, and the second with `cinoptions=s,fs, {s`. For both examples, option `shiftwidth` has the value 4 (four columns).

```
18
19 while (condition)
20 {
21     if (someothercondition)
22     {
23         printf("looks like I've got both conditions!\n");
24     }
25 }
26
```

Figure 11-14. Results of `:set cinoptions=s,f0,{0` *(WSL Ubuntu Linux terminal, color scheme: zellner)*

```
26
27 while (condition)
28     {
29     if (someothercondition)
30         {
31         printf("looks like I've got both conditions!\n");
32         }
33     }
34
```

Figure 11-15. Results of `:set cinoptions=s,fs,{s` *(WSL Ubuntu Linux terminal, color scheme: zellner)*

} *n*

Use this setting to define a closing brace's offset from its matching brace. The default is zero (aligned with the matching brace).

^ *n*

Add *n* to the current indentation inside a set of braces ({…}) if the opening brace is in column one.

: *n,* = *n,* b *n*

These three control indentation in `case` statements. With `:`, Vim indents a `case` label *n* characters from the position of its corresponding `switch` statement. The default is one shiftwidth.

The `=` setting defines the offset for lines of code from their corresponding `case` label. The default is to indent statements one shiftwidth.

The `b` setting defines where to place `break` statements. The default (zero) aligns `break` with the other statements within the corresponding `case` block. Any nonzero value aligns the `break` with its corresponding `case` label.

) *n,* * *n*

These two settings tell Vim how many lines to scan to find unclosed parentheses (default is 20 lines) and unclosed comments (default is 30 lines), respectively.

Ostensibly, these last two settings limit how hard Vim has to work to look for matches. With today's powerful computers, you should consider increasing these values to assure more complete scope management to match comments and parentheses. Try doubling them to 40 and 60, respectively, as a starting point.

indentexpr

`indentexpr`, if defined, overrides `cindent` so that you can define indentation rules and tailor them exactly to your language editing needs.

`indentexpr` defines an expression to be evaluated each time a new line is created in a file. This expression resolves to an integer that Vim uses as the indentation of the new line.

In addition, the option `indentkeys` can define useful keywords in the same way that `cinkeys` keywords define lines after which indentation is reevaluated.

The bad news is that it is a nontrivial project to write customized indentation rules from scratch for any language. The good news is that it's likely that the work is already done. Look in the *$VIMRUNTIME/indent* directory to see whether your

favorite language is represented. A quick peek today (version 8.2) reveals more than 120 indent files.

The most common programming languages are represented, including *ada*, *awk*, *docbook* (the indent file is named *docbk*), *eiffel*, *fortran*, *html*, *java*, *lisp*, *pascal*, *perl*, *php*, *python*, *ruby*, *scheme*, *sh*, *sql*, and *zsh*. There is even an indent file defined for `xinetd`!

You can tell Vim to automatically detect your file type and load the indent file by putting the command `filetype indent on` in your *.vimrc* file. Now Vim will try to detect what file type you are editing and load a corresponding *indent* definition file for you. If the indent rules do not fulfill your needs—for example, if they indent in some unfamiliar or unwanted fashion—turn the definitions off with the command `:filetype indent off`.

We encourage power users to explore and learn from the indent definition files that come with Vim. And if you develop new definition files or improvements to existing ones, we encourage you to submit them to *vim.org* for possible addition to the Vim package.

A Final Word on Indentation

Before ending our discussion, it's worth noting the following points about working with automatic indenting:

When automatic indenting isn't
> Any time you act on a line in an editing session with automatic indenting and you change that line's indentation manually, Vim flags that line and will no longer try to automatically define its indentation.

Copy and paste
> When you paste text into your file where automatic indenting is turned on, Vim considers this regular input and applies all automatic indentation rules. In most cases, this is probably not what you intend. Any indentation in pasted text is tacked on to applied indentation rules. Typically the result is text that progressively skews to the right side of the screen, with large indentation and no corresponding retreat to the left side.

> To avoid this awkward situation and to paste text intact without side effects, set Vim's `paste` option before adding the imported text. `paste` comprehensively reconfigures all of Vim's automatic features to faithfully incorporate pasted text. To return to automatic mode, simply reset the `paste` option with the command `:set nopaste`.

Keyword and Dictionary Word Completion

Vim offers a comprehensive suite of *insertion completion* capabilities. From programming language–specific keywords to filenames, dictionary words, and even entire lines, Vim knows how to offer possible completions to partially entered text. Not only that, but Vim abstracts the semantics of dictionary-based completion to include completions based on synonyms for the completed word from a thesaurus!

In this section we look at the different completion methods, their syntaxes, and descriptions of how they work with examples. The methods of completion include:

- Whole line
- Current file keywords
- `dictionary` option keywords
- `thesaurus` option keywords
- Current and *included* file keywords
- Tags (as in `ctags`)
- Filenames
- Macros
- Vim command line
- User-defined
- Omni
- Spelling suggestions
- `complete` option keywords

Except for `complete` keywords, all completion commands start with CTRL-X. The second key specifically defines the type of completion Vim attempts. For example, the command to autocomplete filenames is CTRL-X CTRL-F. Not all the commands are so mnemonic, unfortunately. Vim uses unmapped keys, which allows you to shorten most of these commands to just the second keystroke by mapping the commands appropriately. For instance, you can map CTRL-X CTRL-N to just CTRL-N.

All completion methods have virtually identical behavior: they cycle through a list of candidate completions as you retype the second keystroke. Thus, if you choose filename autocompletion through CTRL-X CTRL-F and you don't get the right word on the first try, you can repeatedly press CTRL-F to see the other options. Additionally, if you press CTRL-N (for "next"), you move forward through the possibilities, whereas CTRL-P (for "previous") moves backward.

Let's look at some of these autocompletion methods with examples and consider how they might be useful.

Insertion Completion Commands

These methods (used in insert mode) range in function from simply looking for words in your current file to spanning the range of function, variable, macro, and other names throughout an entire suite of code. The final method combines features of the others for a nice compromise between power and sophistication.

> You may want to find your favorite completion method and map it to a single easy-to-use key. One of us maps his to the TAB key:
>
> ```
> :imap Tab <C-P>
> ```
>
> This may sacrifice the ability to insert tabs easily, but it allows him to use the same key as used for completion in command-line environments such as DOS and the shell (xterm, konsole, etc.).
>
> Remember, you can always insert a tab by quoting it with CTRL-V. Mapping to the TAB key also corresponds to the normal completion key in Vim's ex command mode.

The following subsections describe the numerous ways Vim lets you do completion.

Completing whole lines

You complete whole lines by typing CTRL-X CTRL-L. The method looks backward in the current file for a line matching the characters you've typed. We'll try an example to give you a sense of how completion works.

Consider a file that contains terminal, or console, definitions that characterize the features of terminals and how to manipulate them. Say your screen resembles Figure 11-16.


```
341 # implementing some ANSI subset can use many of them.
342 # This terminal is widely used in our company...
343 ansi+local1|ANSI normal-mode cursor-keys,
344         cub1=\E[D, cud1=\E[B, cuf1=\E[C, cuu1=\E[A
```

Figure 11-16. Example of completion by line (Linux gvim, color scheme: zellner)

Note the highlighted line containing "This terminal is widely used in our company…" You need this line in many places as you mark terminals as "widely used" for your company. Simply type enough of the line to make it unique, or close to unique, and then type CTRL-X CTRL-L. Thus, Figure 11-17 contains the partial input line:

```
# Thi
```

Figure 11-17. Partially typed line waiting for completion (Linux `gvim`, color scheme: zellner)

CTRL-X CTRL-L causes Vim to show a set of possible completions for the line, based on lines previously entered in the file. The list of completions is shown in Figure 11-18.

Figure 11-18. After typing CTRL-X CTRL-L *(Linux `gvim`, color scheme: zellner)*

It is hard to see in grayscale (in the printed book), but the screen offers a colored pop-up window containing multiple occurrences of lines matching the beginning of our partial line. Also displayed, but not visible in the screenshot, is information describing where the match is found. This method uses the `complete` option to define the scope for searching for matches. Scope is discussed in detail in the last method of this section.

The pop-up[5] list highlights selections as you move forward (CTRL-N) or backward (CTRL-P) through the list. You may also use the arrow keys to move up and down through the list. Press ENTER to select your match. If you do not want any of the choices in the list, type CTRL-E to halt the match method without substituting any text. Your cursor returns to its original position on the same partial input.

Figure 11-19 shows the results after we select the desired option from the list.

---

5 The pop-up is in `gvim`; Vim behaves slightly differently.

```
954 # This entry is good for the 1.2.13 or later version of the Linux console.
955 # This terminal is widely used in our company...|
956 linux-basic█linux console,
```

Figure 11-19. After typing CTRL-X CTRL-L *and selecting the matching line (Linux* gvim, *color scheme: zellner)*

Completion by keyword in file

CTRL-X CTRL-N searches forward through the current file for keywords matching the keyword in front of the cursor. Once you enter those keystrokes, you can use CTRL-N and CTRL-P to search forward or backward, respectively. Press ENTER to select a match. You may also use the arrow keys to go up and down in the list.

> Note that "keyword" is loosely defined. While it may match keywords programmers are familiar with, it can really match any word in the file. Words are defined as a contiguous set of characters in the iskeyword option. The iskeyword defaults are pretty sane, but you can redefine the option if you want to include or leave out some punctuation. Characters in iskeyword can be specified either directly (such as a-z) or through their corresponding ASCII codes (such as using 97-122 to represent a-z).
>
> For instance, the defaults allow an underscore as part of a word but consider a period or hyphen to be a delimiter. This works fine for C-like languages, but it may not be the best choice for other environments.

Completion by dictionary

CTRL-X CTRL-K searches forward through the files defined by the dictionary option for keywords matching the keyword in front of the cursor.

The default setup leaves the dictionary option undefined. There are common places to find dictionary files, and you can define your own. The most common dictionary files are:

- */usr/share/dict/words* (Cygwin on MS-Windows, Ubuntu GNU/Linux)
- */usr/share/dict/web2* (FreeBSD)
- *$HOME/.mydict* (personal list of dictionary words)

Completion by thesaurus

CTRL-X CTRL-T searches forward through the files defined by the thesaurus option for keywords that match the keyword in front of the cursor.

This method offers an interesting twist. When Vim finds a match, if the line in the thesaurus file contains more than one word, Vim includes all the words in the list of completion candidates.

Ostensibly (and implied by the option's name), this method provides synonyms but allows *you* to define your own standard. Consider the example file containing these lines:[6]

```
fun,enjoyable,desirable
funny,hilarious,lol,rotfl,lmao
retrieve,getchar,getcwd,getdirentries,getenv,getgrent,getgrgid,...
```

The first two lines are typical English-language synonyms (matching "fun" and "funny," respectively), while the third line might be useful for C programmers who regularly insert function names that begin with `get`. The synonym we use for these functions is "retrieve."

In real life, we'd separate the English-language thesaurus from the C-language one, because Vim can search multiple thesauruses.

In input mode, type the word `fun` and then CTRL-X CTRL-T . Figure 11-20 shows the resulting pop-up in `gvim`.

Figure 11-20. Thesaurus completion of "fun" (WSL Ubuntu Linux, color scheme: zellner)

Notice the following:

- Vim matches *any* word it can find in a thesaurus entry, not just the first word of each line in the thesaurus file.
- Vim includes candidate words from all lines in the thesaurus that have a match with the keyword in front of the cursor. Thus, in this case, it finds the matches for both "fun" and "funny."

---

6 Note that the words in each line of synonyms are comma-separated. To include a comma in a word, quote it with a backslash.

Another interesting and perhaps unanticipated behavior of `thesau rus` is that the match can be on words on a line in the thesaurus file *other than* the first word. For instance, in the line from the previous example file:

```
funny hilarious lol rotfl lmao
```

If you type `hilar` and complete it, Vim will include in the list all words from `hilarious` on that line, i.e., "hilarious," "lol," "rotfl," and "lmao." Funny!

Did you notice the extra information in the list of candidates for completion? You can get information about where Vim found the match in the pop-up menu by adding the value `preview` to the `completeopt` option.

Now consider an example, using the same file as before, in which you type the partial word `retrie`. This matches "retrieve," a synonym we like as a mnemonic for "getting" stuff, and we include all "get" function names as synonyms. Now CTRL-X CTRL-T gives us the pop-up menu (in `gvim`) of all of our functions as choices for completion. See Figure 11-21.

Figure 11-21. Thesaurus completion of string `retrie` *(WSL Ubuntu Linux, color scheme: zellner)*

As with other completion methods, press ENTER to select the match.

> Thesaurus completion should not be confused with spellchecking, another nice Vim feature. Visit the section "Spell It! (i-t)" on page 315 for a discussion of Vim's spellchecking.

Completion by keyword in the current file and in included files

This feature is of use to C and C++ programmers, where `#include` files are used a lot. CTRL-X CTRL-I searches forward through the current file and included files for

keywords matching the keyword in front of the cursor. This method differs from the "search current file" method (CTRL-X CTRL-P) in that Vim inspects the current file for *include* file references and searches those files, too.

Vim uses the value in `include` to detect lines referencing *include* files. The default is a pattern telling Vim to find lines matching the standard C construct:

```
# include <somefile.h>
```

In this case, Vim would find matches in the file *somefile.h* in the standard *include* file directories on the system. Vim also uses the `path` option as a list of directories to search for the included files.

Completion by tag

CTRL-X CTRL-] searches forward through the current file and included files for keywords matching *tags*. See the section for a discussion of tags.

Completion by filename

CTRL-X CTRL-F searches for filenames matching the keyword in front of the cursor. Note that this causes Vim to complete the keyword with the *name of the file*, not with words found in files.

> As of Vim 8.2, Vim searches *only* in the current directory for files with possible filename matches. This is in contrast to many Vim features that use the `path` option to look for files. The Vim documentation hints that this behavior is temporary by pointing out that `path` isn't used "yet." However, this has been the case for over a decade…

Completion by macro and definition names

CTRL-X CTRL-D searches forward through the current file and included files for macro names and definitions defined by the `#define` directive.

Completion method with Vim commands

This method, invoked through CTRL-X CTRL-V, is meant for use on the Vim command line and tries to guess the best completions for words. This context is provided to assist users developing Vim scripts.

Completion by user functions

This method, invoked through CTRL-X CTRL-U, lets you define the completion method with your own function. Vim uses the function pointed to by the option

`completefunc` to make the completion. Refer to Chapter 12 for discussions about scripting and writing Vim functions.

Completion by omni function

This method, invoked through CTRL-X CTRL-O, uses user-defined functions much like the previous user function method. The significant difference is that this method expects the functions to be file type–specific and hence to be determined and loaded as a file is loaded. Omni completion files are already available for C, CSS, HTML, JavaScript, PHP, Python, Ruby, SQL, and XML.

Completion for spelling correction

This method is invoked through CTRL-X CTRL-S. The word in front of the cursor is used as the base word for which Vim offers candidates for completion. If the word appears to be badly spelled, Vim suggests "more correct" spellings.

Completion with the complete option

This is the most generic option, invoked through CTRL-N, and lets you combine all the other searches into one search. For many users, this may be the most satisfactory because it requires little understanding of the nuances of the more specific methods.

You define where and how this completion acts by setting the comma-separated list of available sources in the `complete` option. Each available source is (usually) denoted by a single character. The choices include:

. *(period)*
  Search the current buffer.

w

  Search buffers in other windows (within the screen containing your Vim session).

b

  Search other loaded buffers in the buffer list (which might not be visible in any Vim windows).

u

  Search the unloaded buffers in the buffer list.

U

  Search the buffers *not* in the buffer list.

k

  Search the dictionary files (listed in the `dictionary` option).

kspell
    Use the current spellchecking scheme (this is the only option that is not a single character).

s

    Search the thesaurus files (listed in the `thesaurus` option).

i

    Search the current and included files.

d

    Search the current and included files for defined macros.

t,]
    Search for tag completion.

Some Final Comments on Vim Autocompletion

We've covered a lot of material related to autocompletion, but there's lots more. The autocompletion methods yield great returns for the time you invest in mastering their use. If you edit a *lot*, and if there's *any* notion or context of text to be completed, find the method best suited to that and learn it.

One final tip: combinations with two keystrokes (more if you are a typical Unix user and count key combinations as "more than one") can be error-prone, especially given that they are combinations with the CTRL key. If you think you'd use autocompletion heavily, consider mapping your favorite autocompletion to just one keystroke or key combination. Large numbers of autocompletion commands abbreviated to half the length offer that much more efficiency.

The following example shows why we find this customization so valuable. One of us maps the TAB key to generic keyword matching, as mentioned earlier. While editing this book using DocBook XML tags (for the seventh edition), your author typed (using a conservative `grep` of the files) "emphasis" more than 1,200 times! Using keyword completion, he knew that the partial "emph" always matched one choice, the "emphasis" tag he wanted. Thus, for each occurrence of this word, he saved at least three keystrokes (assuming perfect typing for the three initial letters), giving him a total savings of at least 3,600 keystrokes!

Here's another way to measure the efficiency of this method: your author already knows that he types about four characters per second, thus gaining a savings in typing for *one keyword alone* of 3,600 divided by 4, or *15 minutes* saved. For the same DocBook files, he completed another 20 to 30 keywords in the same fashion. The savings in time accrue quickly!

Tag Stacking

Tag stacking is described earlier, in the section "Tag stacks" on page 151.

Besides moving back and forth among the tags you search for, you can choose among multiple matching tags. You can also do tag selection and window splitting with one command. The Vim `ex` mode commands for working with tags are provided in Table 11-1.

Table 11-1. Vim tag commands

Command	Function
ta[g][!] [*tagstring*]	Edit the file containing *tagstring* as defined in the *tags* file. The `!` forces Vim to switch to the new file if the current buffer has been modified but not saved. The file may or may not be written out, depending on the setting of the `autowrite` option.
[*count*]ta[g][!]	Jump to the *count*th newer entry in the tag stack.
[*count*]po[p][!]	Pop a cursor position off the stack, restoring the cursor to its previous position. If supplied, go to the *count*th older entry.
tags	Display the contents of the tag stack.
ts[elect][!] [*tagstring*]	List the tags that match *tagstring*, using the information in the *tags* file(s). If no *tagstring* is given, the last tag name from the tag stack is used.
sts[elect][!] [*tagstring*]	Like `:tselect`, but split the window for the selected tag.
[*count*]tn[ext][!]	Jump to the *count*th next matching tag (default is one).
[*count*]tp[revious][!]	Jump to the *count*th previous matching tag (default is one).
[*count*]tN[ext][!]	
[*count*]tr[ewind][!]	Jump to the first matching tag. With *count*, jump to the *count*th matching tag.
tl[ast][!]	Jump to the last matching tag.

Normally, Vim shows you which matching tag out of how many it has jumped to. For example:

```
tag 1 of >3
```

It uses a greater-than sign (>) to indicate that it has not yet tried all the matches. You can use `:tnext` or `:tlast` to try more matches. If this message is not displayed because of some other message, use `:0tn` to see it.

Here is the output of the `:tags` command, with the current location marked with a greater-than sign (>):

```
  # TO tag      FROM line in file
  1  1 main           1 harddisk2:text/vim/test
> 2  2 FuncA         58 -current-
  3  1 FuncC        357 harddisk2:text/vim/src/amiga.c
```

The :tselect command lets you pick from more than one matching tag. The "priority" (pri field) indicates the quality of the match (global versus static, exact case versus case-independent, etc.); this is described more fully in the Vim documentation:

```
nr pri kind tag               file ~
  1 F   f    mch_delay         os_amiga.c
                mch_delay(msec, ignoreinput)
> 2 F   f    mch_delay         os_msdos.c
                mch_delay(msec, ignoreinput)
  3 F   f    mch_delay         os_unix.c
                mch_delay(msec, ignoreinput)
Enter nr of choice (<CR> to abort):
```

The :tag and :tselect commands can be given an argument that starts with /. In that case, the command uses it as a regular expression, and Vim will find all the tags that match the given regular expression.

For example, :tag /normal will find the macro NORMAL, the function normal_cmd, and so on. Use :tselect /normal and enter the number of the tag you want.

The Vim command mode commands are described in Table 11-2. Besides using the keyboard, you can also use the mouse if mouse support is enabled in your version of Vim.

Table 11-2. Vim command mode tag commands

Command	Function
^]	Look up the location of the identifier under the cursor in the *tags* file, and move to that location. The current location is automatically pushed onto the tag stack.
g <LeftMouse> CTRL-<LeftMouse> ^T	Return to the previous location in the tag stack, i.e., pop off one element. A preceding count specifies how many elements to pop off the stack.

The Vim options that affect tag searching are described in Table 11-3.

Table 11-3. Vim options for tag management

Option	Function
taglength, tl	Controls the number of significant characters in a tag that is to be looked up. The default value of zero indicates that all characters are significant.
tags	The value is a list of filenames in which to look for tags. As a special case, if a filename starts with ./, the dot is replaced with the directory part of the current file's pathname, making it possible to use *tags* files in a different directory. The default value is ./tags,tags.
tagrelative	When set to true (the default) and using a *tags* file in another directory, filenames in that *tags* file are considered to be relative to the directory where the *tags* file is.

Vim can use Emacs-style *etags* files, but this is only for backward compatibility; the format is not documented in the Vim documentation, nor is the use of *etags* files encouraged.

Finally, Vim also looks up the entire word containing the cursor, not just the part of the word from the cursor location forward.

Syntax Highlighting

One of Vim's strongest enhancements to vi is its syntax highlighting. Vim's syntax formatting relies heavily on the use of color, but it also degrades gracefully on screens that do not support color. In this section we discuss three topics: getting started, customizing, and rolling your own. Syntax highlighting for Vim contains features that go beyond the scope of this book, so we focus on providing enough information to get you familiar with it and enable you to extend it to fit your needs.

Because the impact of Vim's syntax highlighting is most dramatic in color, and this book isn't in color, we strongly encourage you to try syntax highlighting to fully appreciate the power of color in defining context. We never met a user who tried it and did not continue to always use it.[7]

Getting Started

Displaying a file's syntax highlighting is simple. Just issue the command:

```
:syntax enable
```

If all is well, when you edit a file with a formal syntax, such as a programming language, you should see text in various colors, all determined by context and syntax. If nothing changed, try turning syntax highlighting on:

```
:syntax on
```

When syntax on Isn't Enough

Enabling syntax highlighting should be enough by itself, but we have encountered situations in which more work was required.

If you still see no syntax highlights, Vim may not know what your file type is and thus not understand which syntax is appropriate. There are a number of reasons this happens.

---

7 Well, with the exception of one of our reviewers!

For example, if you create a new file and don't use a recognized suffix, or any suffix at all, Vim cannot determine the file type because the file is new and therefore empty. For instance, we often write shell scripts without any *.sh* suffix. Each new shell script begins its editing life without syntax highlighting. Fortunately, once the file contains code, Vim knows how to figure out the file type, and syntax highlighting works as expected.

It's also possible (though not likely) that Vim doesn't have a syntax description for your file type. This is very rare, and usually you just need to specify a file type explicitly, because someone has already written a syntax file for the language. Unfortunately, creating one from scratch is a complex undertaking, although we give you some tips later in this chapter.

You can force Vim to use a particular syntax highlighting style by setting the syntax manually from the `ex` command line. When starting a new shell script, for instance, we always define the syntax with:

```
:set syntax=sh
```

The section "Dynamic File Type Configuration Through Scripting" on page 300 shows a clever if rather roundabout way to avoid this step.

When you enable syntax highlighting, Vim sets it up by going through a checklist. Without getting mired in too many technical details, we'll just say that Vim ultimately determines your file type, finds the appropriate syntax definition file, and loads it for you. The standard location for syntax files is the *$VIMRUNTIME/syntax* directory.

To get a sense of the comprehensive coverage of syntax definitions, the Vim syntax file directory contains *almost 500 syntax files*. Available syntaxes span the gamut from languages (C, Java, HTML) to content (calendar) to well-known configuration files (*fstab*, *xinetd*, *crontab*). If Vim doesn't recognize your file type, try looking in the *$VIMRUNTIME/syntax* directory for a syntax file that closely matches yours.

Customization

Once you start using syntax highlighting, you may find that some of the colors do not work for you. They may be difficult to see or may just not suit your taste. Vim has a few ways to customize and tune colors.

Here are some things to try before taking more drastic measures (e.g., writing your own syntax description, as described in the next section) to make syntax highlighting work for you.

Two of the most common and dramatic symptoms of syntax highlighting gone amok are:

- Bad contrast, with colors too similar and hard to see distinctly as different from each other

- Too many or too varied colors, which gives a harsh look to the text

Although these are subjective deficiencies, it's nice that Vim lets you make corrections. Two commands, `colorscheme` and `highlight`, and one option, `background`, can probably bring the colors to a satisfactory balance.

There are a few other commands and options with which you can customize your syntax highlighting. After a brief introduction to *syntax groups*, we talk about these commands and options in the following sections, with an emphasis on the three just mentioned.

Syntax groups

Vim classifies different types of text into groups. These groups each receive color and highlight definitions. Additionally, Vim allows groups of groups. You can address definitions at different levels. If you assign a definition to a group containing subgroups, unless otherwise defined, each subgroup inherits the parent group's definitions.

Some high-level groups for syntax highlighting include:

Comment
    Comments specific to the programming language, e.g.:

```
// I am both a C++ and a JavaScript comment
```

Constant
    Any constant—e.g., `TRUE`.

Identifier
    Variable and function names.

Type
    Declarations, such as `int` and `struct` in C.

Special
    Special characters, such as delimiters.

Taking the *special* group from the previous list, we can look at an example of subgroups:

- SpecialChar

- Tag

- Delimiter

- SpecialComment
- Debug

With a basic understanding of syntax highlighting, groups, and subgroups, we now know enough to modify syntax highlighting to suit our tastes.

The colorscheme command

This command changes colors for different syntax highlights such as comments, keywords, or strings by redefining these syntax groups. Vim ships with the following color scheme choices:[8]

```
blue      delek    evening  murphy     ron    torte
darkblue  desert   koehler  pablo      shine  zellner
default   elflord  morning  peachpuff  slate
```

These files are in the directory *$VIMRUNTIME/colors*. You can activate any one of them with:

```
:colorscheme scheme_name
```

In Vim and `gvim`, you can quickly cycle through the different schemes this way: type the partial command `:color`, press TAB to start command completion, press the space bar, and then repeatedly press TAB to cycle through the different choices.

In `gvim`, the choice is even easier. Click on the Edit menu, move the mouse over the Colorscheme submenu, and tear off the menu. Now you can look at all the choices by clicking each button.

There are *many* color schemes available, all contributed by Vim's user community. You may be interested in the GitHub repository, from which almost a thousand color schemes are available for download.

Setting the background option

When Vim sets colors, it first tries to determine what kind of background color your screen has. Vim has just two categories for background: dark or light. Based on Vim's determination, it sets colors differently for each, with the end result hopefully being a set of colors that works well with that background (i.e., one with good contrast and color compatibility). Although Vim does try very hard, a correct assessment is tricky, and an assignment to dark or light is subjective. Sometimes the contrasts render the session uncomfortable to view, and sometimes they are unreadable.

---

8 We've noticed that some instances of Vim may have a slightly different set of default color schemes.

So if the colors don't look good, try explicitly choosing a `background` setting. Make sure first to identify the setting:

```
:set background?
```

so that you know that you are changing the setting. Then issue a command such as:

```
:set background=dark
```

Use the `background` option in tandem with the `colorscheme` command to fine-tune your screen colors. These two together can usually produce a satisfactory color palette that is comfortable to view.

The highlight command

Vim's `highlight` command lets you manipulate different groups and control how they are highlighted in your editing session. This command is powerful. You can inspect settings for various groups either as a list or by requesting specific group highlight information. For example:

```
:highlight comment
```

returns Figure 11-22. The first field lists the highlight name (in this example, "Comment"). The second field always displays the string "xxx" as it appears as defined by the highlight definitions in the terminal or GUI.

```
:hi comment
Comment            xxx term=bold ctermfg=12 guifg=Red
```

Figure 11-22. Highlight for comments (WSL Ubuntu Linux terminal, color scheme: zellner)

The output shows how comments in this file will appear. The `xxx` is dark gray on the printed page, but on the screen it's red.[9] The `term=bold` output means that on a terminal incapable of color, `Comments` will be shown in bold. `ctermfg=12` means that on a color terminal, such as an `xterm` on a color monitor, the foreground color for `Comments` will be the matching DOS color blue. Finally, `guifg=Red` means the GUI interface will display `Comments` with the foreground color red.

---

9 To view Figures 11-22 through 11-27 in color, please visit the O'Reilly website.

The DOS color scheme is a more restricted set of colors than in modern GUI sets. In this scheme, there are only eight colors: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, and `white`. Each of them can be set for text foreground or background and optionally can be defined as "bright," a brighter color on the screen. Vim uses analogous mappings for defining text colors in non-GUI windows, e.g., in `xterms`.

GUI windows offer virtually unlimited color definitions. Vim lets you define some colors with common names such as `Blue`, but you can also define these colors with red, green, and blue values. The format is `#rrggbb` where the `#` is literal, and `rr`, `gg`, and `bb` are hexadecimal numbers representing the level of each color. For example, red could be defined with `#ff0000`.

Use the `highlight` command to change settings for groups whose colors you don't like. For example, we can find that identifiers in this file are dark cyan for our GUI interface, as shown in the output in Figure 11-23:

```
:highlight identifier
```


Figure 11-23. Highlight for identifiers

We can redefine the color for identifiers with the command:

```
:highlight identifiers guifg=red
```

Now all identifiers on the screen are (a rather ugly) red. This kind of customization is fairly inflexible: it applies to all file types and does not adapt to different backgrounds or color schemes.

To see how many highlight definitions exist and what their values are, again use `highlight`:

```
:highlight
```

Figure 11-24 shows a small sample of the results from the `highlight` command.

```
Identifier     xxx term=underline ctermfg=9 guifg=red
Statement      xxx term=bold ctermfg=4 guifg=Brown
PreProc        xxx term=underline ctermfg=13 guifg=Purple
Type           xxx term=underline ctermfg=9 guifg=Blue
Underlined     xxx term=underline cterm=underline ctermfg=5 gui=underline guifg=SlateBlue
```

Figure 11-24. Partial results from the `:highlight` *command (WSL Ubuntu Linux terminal, color scheme: zellner)*

Note how some lines contain full definitions (listing `term`, `ctermfg`, and so on), whereas others receive their attributes from parent groups (e.g., `String` links back to `Constant`).

Overriding syntax files

In the previous section, we learned how to define syntax group attributes for all instances of a group. Suppose you want to change a group for only one or a few syntax definitions. Vim lets you do this with the *after* directory. This is a directory in which you can create any number of *after* syntax files that Vim will execute after the normal syntax file.

To do this, simply include highlight commands (or any processing commands—the notion of "after" processing is generic) in the specific file in a directory named *after* that is included in the `runtimepath` option. Now when Vim sets up syntax highlighting rules for your file type, it will also execute your custom commands in the *after* file.

For example, let's apply a customization to XML files, which use the `xml` syntax. This means Vim loaded syntax definitions from a file in the syntax directory named *xml.vim*. As in the previous example, we want to define identifiers always to be red. So we create our own file named *xml.vim* in a directory named *~/.vim/after/syntax*. In our *xml.vim* file we put the following line:

```
highlight identifier ctermfg=red guifg=red
```

Before this customization works, we must ensure that *~/.vim/after/syntax* is in the `runtimepath` path:

```
:set runtimepath+=~/.vim/after/syntax        In our .vimrc file
```

To make the change permanent, of course, the line should go in our *.vimrc* file.

Now, whenever Vim loads syntax definitions for `xml`, it will override the definitions for `identifier` with our own customization.

Rolling Your Own

With the building blocks from the previous sections, we now have enough knowledge to write our own syntax files, simple as they might be. There are still many facets to learn before we can fully develop a syntax file.

We will incrementally build our own syntax file. Because syntax definitions can be extremely complex, let's consider something simple enough to be easily grasped, but complex enough to show its potential power.

Consider an excerpt from a generated Latin file, *loremipsum.latin*:

```
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin eget
tellus. Suspendisse ac magna at elit pulvinar aliquam. Pellentesque
iaculis augue sit amet massa. Aliquam erat volutpat. Donec et dui at
massa aliquet molestie. Ut vel augue id tellus hendrerit porta. Quisque
condimentum tempor arcu. Aenean pretium suscipit felis. Curabitur semper
eleifend lectus. Praesent vitae sapien. Ut ornare tempus mauris. Quisque
ornare sapien congue tortor.

In dui. Nam adipiscing ligula at lorem. Vestibulum gravida ipsum iaculis
justo. Integer a ipsum ac est cursus gravida. Etiam eu turpis. Nam laoreet
ligula mollis diam. In aliquam semper nisi. Nunc tristique tellus eu
erat. Ut purus. Nulla venenatis pede ac erat.

...
```

You create a new syntax file by creating a new file of that syntax name, in this case `latin`. Its corresponding Vim file is *latin.vim*, which you can create in your personal Vim runtime directory, *$HOME/.vim/syntax*. Then start your syntax definition simply by creating some keywords with the `syntax keyword` command. Choosing *lorem*, *dolor*, *nulla*, and *lectus* as keywords, you can inaugurate the syntax file with the line:

```
syntax keyword identifier lorem dolor nulla lectus
```

There still isn't any syntax highlighting when you edit *loremipsum.latin*. More work needs to be done before highlighting is automatic. But for the time being, activate the syntax with the command:

```
:set syntax=latin
```

The text should now look something like Figure 11-25.

Figure 11-25. Latin file with keywords defined

On the screen, the text words are black with the keywords in red. In the printed book, it's harder to distinguish; the keywords are dark gray instead of black.

You may have noticed that the first occurrence of *Lorem* isn't highlighted. By default, syntax keywords are case sensitive. Add the following line at the top of the syntax file:

```
:syntax case ignore
```

and you should now see *Lorem* included as a highlighted keyword.

Before we try this again, let's make it all work automatically. After Vim tries to detect any file type, it optionally checks for other definitions, or even overriding definitions (which are not recommended), in a directory named *ftdetect* in your `runtimepath`. Therefore, create that directory under *$HOME/.vim* and create a file in it named *latin.vim* containing this single line:

```
au BufRead,BufNewFile *.latin set filetype=latin
```

This tells Vim that any files with the suffix *.latin* are `latin` files, and therefore that Vim should execute the syntax file in *$HOME/.vim/syntax/latin.vim* when displaying them.

Now when you edit *loremipsum.latin*, you see Figure 11-26.


```
 1 Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin eget
 2 tellus. Suspendisse ac magna at elit pulvinar aliquam. Pellentesque
 3 iaculis augue sit amet massa. Aliquam erat volutpat. Donec et dui at
 4 massa aliquet molestie. Ut vel augue id tellus hendrerit porta. Quisque
 5 condimentum tempor arcu. Aenean pretium suscipit felis. Curabitur semper
 6 eleifend lectus. Praesent vitae sapien. Ut ornare tempus mauris. Quisque
 7 ornare sapien congue tortor.
 8
 9 In dui. Nam adipiscing ligula at lorem. Vestibulum gravida ipsum iaculis
10 justo. Integer a ipsum ac est cursus gravida. Etiam eu turpis. Nam laoreet
11 ligula mollis diam. In aliquam semper nisi. Nunc tristique tellus eu
12 erat. Ut purus. Nulla venenatis pede ac erat.
12
```

Figure 11-26. Latin file with keywords defined, ignoring case (WSL Ubuntu Linux terminal, color scheme: zellner)

First, notice that the syntax was active right away, as Vim correctly detected your new syntax file type, *latin*. And keywords now match without any sensitivity to case.

For some more interesting extensions, define a `match` and assign it to group `Comment`. The `match` method uses a regular expression to define what is highlighted. For example, we will define all words beginning with `s` and ending with `t` to be `Comment` syntax (remember, this is just an example!). Our regular expression is: `\<s[^\t ]*t\>`. We also will define a region and highlight it as a `Number`. Regions are defined with a `start` and `end` regular expression.

Our region begins with `Suspendisse` and ends with `sapien\.`. To add even more of a twist, we decide that the keyword `lectus` is *contained* within our region. Our *latin.vim* syntax file now looks like:

```
syntax case ignore
syntax keyword identifier lorem dolor nulla lectus
syntax keyword identifier lectus  contained
syntax match comment /\<s[^\t ]*t\>/
syntax region number start=/Suspendisse/ end=/sapien\./ contains=identifier
```

Now when we edit *loremipsum.latin*, we see Figure 11-27.

Figure 11-27. New `latin` syntax highlighting (WSL Ubuntu Linux terminal, color scheme: zellner)

There are several things to notice, which you can see much more easily if you run the example and view the results in color:

- The new match highlights appear. On the first line, *dolor sit* is highlighted in red because it satisfies the regular expression for the `match`.

- The new region highlights appear. The entire section of the paragraph beginning with *Suspendisse* through *sapien.* is highlighted in purple (ick).

- The keywords are still highlighted as before.

- Within the highlighted region, the keyword *lectus* is still highlighted in red because we defined group `identifier` as `contained` and defined our region as `contains identifier`.

This example only begins to tap the rich powers of syntax highlighting. Although this particular example is somewhat useless, we hope that it demonstrates enough to convince you of this feature's power and encourages you to experiment and create your own syntax definitions.

Compiling and Checking Errors with Vim

Vim isn't an IDE, but it tries to make life a little easier for programmers by incorporating compilation into the editing session and providing a quick and easy way to find and correct errors.

Additionally, Vim offers some convenience functions to track and navigate *locations* in your files. We discuss a simple example: the edit-compile-edit cycle using Vim's built-in features and some of its related commands and options, as well as the convenience functions. All of these depend on the same Vim `Quickfix List` window.

As a simple starting point, Vim lets you compile files using `make` each time you change one. Vim uses default behavior to manage the results of your build so that you can easily alternate between editing and compilation. Compilation errors appear in Vim's special `Quickfix List` window, where you can inspect, jump to, and correct errors.

For this discussion we use a little C program that generates Fibonacci numbers.[10] In its correct and compilable form, the code is:

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

int main(int argc, char *argv[])
  {
  /*
   * arg 1: starting value
   * arg 2: second value
   * arg 3: number of entries to print
   *
   */

  if (argc - 1 != 3)
    {
    printf ("Three command line args: (you used %d)\n", argc-1);
    printf ("usage: value 1, value 2, number of entries\n");
    return (1);
    }


  /* count = how many to print */
  int count = atoi(argv[3]);

  /* index = which to print */
  long int index;

  /* first and second passed in on command line */
  long int first, second;
```

---

10 The file is available in the book's GitHub repository; see the section "Accessing the Files" on page 471.

```
/* these get calculated */
long int current, nMinusOne, nMinusTwo;

first  = atoi(argv[1]);
second = atoi(argv[2]);
printf("%i fibonacci numbers with starting values: %li, %li\n", count, first,
    second);
printf("=====================================\n");

/* print the first 2 from the starter values */
printf("%i %04li\n", 1, first);
printf("%i %04li  ratio (golden?) %.3f\n", 2, second, (double) second/first);

nMinusTwo = first;
nMinusOne = second;

for (index=1; index<=count; index++)
  {
  current = nMinusTwo + nMinusOne;
  printf("%li %04li  ratio (golden?) %.3f\n",
        index,
        current,
        (double) current/nMinusOne);
  nMinusTwo =  nMinusOne;
  nMinusOne = current;
  }
}
```

From Vim, compile this program (assuming a filename of *fibonacci.c*) with the command:

```
:make fibonacci
```

By default, Vim passes the `make` command through to the external shell and captures the results in the special `Quickfix List` window. After compiling the previous code, the screen with the `Quickfix List` window looks something like Figure 11-28.

Figure 11-28. `Quickfix List` window after a clean compile (WSL Ubuntu Linux terminal, color scheme: zellner)

If you don't see the QuickFix window (it did not open automatically), you can open it with the Vim ex command `:copen`.

Next, we change enough lines in our program to introduce a healthy number of errors.

Change:

```
long int current, nMinusOne, nMinusTwo;
```

to the following invalid declaration:

```
longish int current, nMinusOne, nMinusTwo;
```

Change:

```
nMinusTwo = first;
nMinusOne = second;
```

to misspelled variables `xfirst` and `xsecond`:

```
nMinusTwo = xfirst;
nMinusOne = xsecond;
```

Change:

```
printf("%d %04li  ratio (golden?) %.3f\n", 2, second, (double) second/first);
```

to this, with missing commas:

```
printf("%d %04li  ratio (golden?) %.3f\n", 2 second (double) second/first);
```

Now recompile the program. Figure 11-29 shows what the QuickFix List window now contains.

Figure 11-29. `Quickfix List` window after a compilation with errors (WSL Ubuntu Linux terminal, color scheme: zellner)

Line 1 of the `Quickfix List` window shows the compilation command executed. If there had been no errors, this would be the only line in the window. But because there are errors, line 3 begins the list of errors and their contexts.

Vim lists all the errors in the `Quickfix List` window and lets you access the code, where errors are indicated in several ways. Vim starts with the convenience behavior by highlighting the first error in the `Quickfix List` window. It then repositions the source file (scrolling if necessary) and places the cursor at the beginning of the source code line corresponding to the error.

As you fix errors, you can navigate to the next error in one of two ways: enter the command `:cnext`, or position the cursor over the error line in the `Quickfix List` window and press ENTER. Again, Vim scrolls the source file if necessary, and positions the cursor at the beginning of the offending source code line.

After you've made changes and are satisfied that you've corrected your errors, you're ready to begin the compile-edit cycle again using the same technique. If you have a standard developer's environment (which is almost always true for Unix/Linux machines), Vim's default behaviors will handle edit-compile-edit as described without any tweaking.

If Vim's defaults don't find a proper compilation program, it has options you can use to define where utilities are located, to let you do your work. The details about programming environments and compilers are outside the scope of this discussion, but we present these Vim options as a starting point in case you need to play with your environment:

`:cnext, :cprevious`
> Commands that move the cursor to *next* and *previous* error locations, as defined in the `Quickfix List` window, respectively.

`:colder, :cnewer`
> Vim remembers the last 10 lists of errors. These commands load the next *older* or next *newer* list of errors in the `QuickFix List` window. Each command takes an optional integer *n* to load the *n*th older or newer error list.

`errorformat`
> An option defining a format that Vim matches to find errors returned from a compile. Vim's built-in documentation gives much more detailed information on how this is defined, but the default almost always works. If you need to tune the option, view its details with:
>
>   :help errorformat

`makeprg`
> An option containing the name of the development environment's `make` or `compile` program.

More Uses for the Quickfix List Window

Vim also lets you build your own list of locations within files, specifying the locations through a `grep`-like syntax. The `QuickFix List` window returns the results you asked for in a format closely resembling the lines returned from the compilation process described earlier. This is done with the `:vimgrep` command, whose syntax is:

```
:vimgrep[!] /pattern/[g][j] file(s)
```

This is essentially a built-in version of the standard *grep*(1) utility. It searches the *files* for lines that match the *pattern* and places the results into the QuickFix window. (See the Vim documentation for the details on the flags and their meanings.)

This feature is useful for such tasks as refactoring. As an example, we composed this manuscript in AsciiDoc. At some point in the composition process we switched the notation for any occurrence of "++vim++" from `++vim++` to `__vim__`.[11] So, each occurrence like:

```
++vim++
```

needed to be changed to:

```
__vim__
```

After executing this command:

```
:vimgrep /++vim++/ *.asciidoc
```

the `Quickfix List` window contained the information shown in Figure 11-30.

```
 1 appd.asciidoc|51 col 26| If the command ++vi++ or ++vim++ doesn't start your editor it is either not
 2 appd.asciidoc|208 col 43| When done, you'll have an executable name ++vim++.  To install
 3 ch01.asciidoc|31 col 1| ++vim++ is the Unix command that invokes the Vim editor for an existing
 4 ch01.asciidoc|318 col 50| file or for a brand new file. The syntax for the ++vim++ command is:
 5 ch02.asciidoc|1562 col 36| |Start Vim, open file if specified|++vim++ __++file++__
 6 ch04.asciidoc|98 col 32| There are other options to the ++vim++ command that can be helpful. You
 7 ch04.asciidoc|142 col 10| Give the ++vim++ command with the option ++$$-c /$$++__++pattern++__
 8 ch08.asciidoc|726 col 1| ++vim++::
 9 ch08.asciidoc|952 col 1| ++VIM++::
10 ch08.asciidoc|965 col 54| If more than one version of Vim exists on a machine, ++VIM++ will
11 ch08.asciidoc|968 col 10| sets the ++VIM++ environment variable to __$$/$$usr$$/$$share$$/$$vim__,
```

Figure 11-30. `Quickfix List` window after `:vimgrep` command (WSL Ubuntu Linux terminal, color scheme: shine)

> Remember to open the QuickFix window with the Vim `ex` command:
>
> ```
> :copen
> ```
>
> Otherwise you will not see the results.

---

11 The notation changed some more later on.

Notice in Figure 11-30 how Vim displays the `:vimgrep` output. On the left are the QuickFix buffer line numbers, which simply indicate how many lines are in the output; they can be turned off with the Vim `ex` command:

```
:set nonumber
```

The `:vimgrep` output comprises three fields separated by the pipe character (`|`). The first field is the name of the file where `vimgrep` matched the pattern. The second field describes the line and column where the matching pattern was found. The third field is the actual text of the matching line.

You can navigate to any of the matches by moving the cursor to the line you are interested in, or you can double-click the line of interest. Vim opens that file in another (split) window and positions the cursor over the first matching pattern character.

In Figure 11-30 the highlighted line (which is a little difficult to read, though we tried to select a color scheme as readable as possible) is a pointer to line 98, column 32, in file *ch04.asciidoc*. See Figure 11-31 for the results after double-clicking that line; you can see how Vim placed the cursor in the correct line and column in *ch04.asciidoc* in the corresponding window.

```
 98 There are other options to the ++vim++ command that can be helpful. You
 99 can open a file directly to a specific line number or pattern. You can
100 also open a file in read-only mode. Another option recovers all changes
101 to a file that you were editing when the system crashed.
102
103 The options described in the following section apply both to ++vi++
104 and to Vim.
105
106 [[vi8-ch-4-sect-2.1]]
107 ==== Advancing to a Specific Place
108
109 When you begin editing an existing file, you can call the file in and
110 then move to the first occurrence of a __pattern__ or to a specific line
ch04.asciidoc    Mon Aug 30 14:53:30 2021
```

Figure 11-31. Vim positioning the cursor in the file, line, and column from the `vimgrep` QuickFix window (WSL Ubuntu Linux terminal, color scheme: shine)

So it was a simple matter to navigate through all occurrences and quickly change to the new values.

This example may seem to solve a problem more easily solved with this simple command:

```
:%s/++vim++/__vim__/g
```

But remember, `vimgrep` is more general and operates against multiple files. This is an example of what `vimgrep` does, not a definitive way to perform this task. In Vim, there are usually many ways to get something done.

Some Final Thoughts on Vim for Writing Programs

We have looked at many powerful features in this chapter. Spend some time mastering these techniques and you'll gain great productivity. If you're a longtime `vi` user, you've already climbed one steep learning curve. The extra effort to learn Vim's additional features is worth a second learning curve.

If you're a programmer, we hope this chapter shows how much Vim offers for your programming tasks. We encourage you to try some of these features and even to extend Vim to suit your own needs. And maybe you will create extensions to give back to the Vim community. Now, go program!

Vim Scripts

Sometimes customization alone isn't enough for your editing environment. Vim lets you define all of your favorite settings in your *.vimrc* file, but maybe you want more dynamic or "just in time" configuration. Vim scripts let you do that.

From inspecting buffer contents to handling unanticipated external factors, Vim's scripting language lets you complete complex tasks and make decisions based on *your* needs.

If you have a Vim configuration file (*.vimrc*, *.gvimrc*, or both), you are already scripting in Vim; you just don't know it. All of the Vim commands and options are valid inputs to scripts. And, as you'd expect, Vim provides all of the standard flow control statements (`if…then…else`, `while`, etc.), variables, and functions typical in any language.

In this chapter, we'll walk through an example and incrementally build up a script. We'll look at simple constructs, use some of Vim's built-in functions, and examine rules you must consider in order to write well-behaved and predictable Vim scripts.

What's Your Favorite Color (Scheme)?

Let's begin with the simplest of configurations. We'll customize our environment to a color scheme *we* prefer. This is simple and uses one of the basics of Vim scripts, the simple Vim command.

Vim ships with 17 customized color schemes.[1] You can choose and activate a color scheme by putting the `colorscheme` command in your *.vimrc* or *.gvimrc* file. A favorite "understated" color scheme of one author is the desert scheme:

---

1 We've heard others report slightly different numbers and default color schemes, but this is pretty close.

```
:colorscheme desert
```

Put a `colorscheme` like that in your configuration file, and now every time you edit with Vim you will see your favorite colors.

So our first script is trivial. What if your tastes for your color scheme are more complex? What if you like more than one color scheme? What if the time of day correlates to your preferences? Vim scripts easily let you do this.

> Choosing an alternate color scheme depending on the time of day may seem trite, but maybe not as much as you may think. Even Google changes the colors and tone of your *iGoogle* home page throughout the day.

Conditional Execution

One of us likes to divide the day into four partitions, each with its own dedicated color scheme:

darkblue
    Midnight to 6 a.m.

morning
    6 a.m. to noon

shine
    Noon to 6 p.m.

evening
    6 p.m. to midnight

We'll build a nested `if…then…else…` block of code for this purpose. There are several different syntaxes you can use for this block. One is more traditional, with an explicitly laid-out syntax:

```
if cond expr
  line of vim code
  another line of vim code
  ...
elseif some secondary cond expr
  code for this case
else
  code that runs if none of the cases apply
endif
```

The `elseif` and `else` blocks are optional, and you can include multiple `elseif` blocks. Vim also allows this more terse and C-like construct:

```
cond ? expr 1 : expr 2
```

Vim checks the condition *cond*. If it's true, *expr 1* executes; otherwise, *expr 2* executes.

Using the strftime() function

Now that we can conditionally execute code, we need to figure out what part of the day it is. Vim has built-in *functions* that return this kind of information. In our case, we use the `strftime()` function. `strftime()` accepts two parameters, the first of which defines the output format of a time string. This format is system dependent and not portable, so you must pay due care when choosing a format. Fortunately, most mainstream formats are common across systems. The second optional parameter is a time measured in seconds since January 1, 1970 (the standard C time representation). This optional parameter defaults to the current time. For our example, we can use the time format `%H`, producing `strftime("%H")`, because the hour of the day is all we need to decide on our color scheme.

Now that we know how to use conditional code, we have the Vim built-in function to give us the information about the time of day with which we choose our matching color scheme. Put this code into your *.vimrc* file:

```
" progressively check higher values... falls out on first "true"
if strftime("%H") < 6
  colorscheme darkblue
  echo "setting colorscheme to darkblue"
elseif strftime("%H") < 12
  colorscheme morning
  echo "setting colorscheme to morning"
elseif strftime("%H") < 18
  colorscheme shine
  echo "setting colorscheme to shine"
else
  colorscheme evening
  echo "setting colorscheme to evening"
endif
```

Notice that we introduce another Vim script command, `echo`. As a convenience, we add this to echo the current scheme to ourselves; it also lets us check that the code actually ran and produced the desired result. The message should appear in Vim's command status window or as a pop-up, depending on where in the startup sequence the `echo` command is encountered.

When we issue the command `colorscheme`, we use the name of the scheme (e.g., `desert`) *without* surrounding quotes, but when we use the `echo` command, we *do* quote the name (`"desert"`). This is an important distinction!

In the case of the `colorscheme` command in our script, we are issuing a direct Vim command, and the parameter for this command is a literal. If we include surrounding quotes, the quotes are interpreted as part of the name of the color scheme by `colorscheme`. This is an error because none of the schemes include quotes in their names.

On the other hand, the `echo` command interpolates words without quotes as expressions (calculations that return values) or functions. Therefore, we need to quote the name of the color scheme we choose.

Variables

If you are a programmer, you probably see a problem with the script we just presented. While it's unlikely to be a big concern in what we are trying to do, we are executing a conditional check of the hour of the day by invoking the `strftime()` function at each conditional point. Technically, we are conditionally checking one thing, but we are evaluating it as an expression multiple times, potentially making a conditional decision on something that changes value mid-execution.

Instead of executing the function each time, let's evaluate it once and store the results in a Vim script *variable*. We can then use the variable as often as we want in our conditional, without incurring the overhead of a function call.

You assign a value to a variable with the `:let` command:

```
:let var = "value"
```

For our purposes, we can define our variable any way we want (context allowing) because we use it only once (though this will change later). For now, we let Vim treat it as global by default. Later we will see that you can use special prefixes to define a variable's scope.

Let's call our variable `currentHour`.[2] By assigning the result from `strftime()` only once, we now have a more efficient script:

```
" progressively check higher values... falls out on first "true"
let currentHour = strftime("%H")
```

---

2 An observation by a technical reviewer, with which we agree: *The variable name `currentHour` is a bit of a misnomer, because its value isn't really an hour. This current name reflects its heritage, but alternatively, we could name the variable for its usage: `colorIndex`. Still, we will leave the script in its original form.*

```
      echo "currentHour is " currentHour
      if currentHour < 6
        colorscheme darkblue
        echo "setting colorscheme to darkblue"
      elseif currentHour < 12
        colorscheme morning
        echo "setting colorscheme to morning"
      elseif currentHour < 18
        colorscheme shine
        echo "setting colorscheme to shine"
      else
        colorscheme evening
        echo "setting colorscheme to evening"
      endif
```

We can clean up the code a little more and get rid of a few lines by introducing a variable named `colorScheme`. This variable holds the value of the color scheme that we determine by time of day. We'll add a capital *S* to distinguish the variable from the name of the `colorscheme` command, but we could have used the exact same letters and it wouldn't matter—Vim can determine from the context what is a command and what is a variable:

```
      " progressively check higher values... falls out on first "true"
      let currentHour = strftime("%H")
      echo "currentHour is " . currentHour
      if currentHour < 6
        let colorScheme ="darkblue"
      elseif currentHour < 12
        let colorScheme = "morning"
      elseif currentHour < 18
        let colorScheme = "shine"
      else
        let colorScheme = "evening"
      endif
      echo "setting color scheme to " . colorScheme
      colorscheme colorScheme
```

Notice the use of the dot (`.`) notation with the `echo` command. This operator concatenates expressions into one string, which `echo` ultimately displays. In this case we concatenate a literal string, `"setting color scheme to "`, and the value assigned to the variable `colorScheme`.

We made an incorrect assumption about executing commands within this script. If you coded along with the example, you already know this. We correct the error in the next section.

The execute Command

So far we have improved how we pick our color scheme, but our last change introduced a slight twist. Initially, we decided to execute a discrete `colorscheme` command

based on time of day. Our last improvement looks correct, but after defining a variable (`colorScheme`) to hold the value of our color scheme, we find that the command:

```
colorscheme colorScheme
```

results in the error shown in Figure 12-1.


```
Message

    currentHour is 10
    setting color scheme tomorning
    Error detected while processing C:\Users\elhan\.vimrc:
    line 122:
    E185: Cannot find color scheme 'colorScheme'
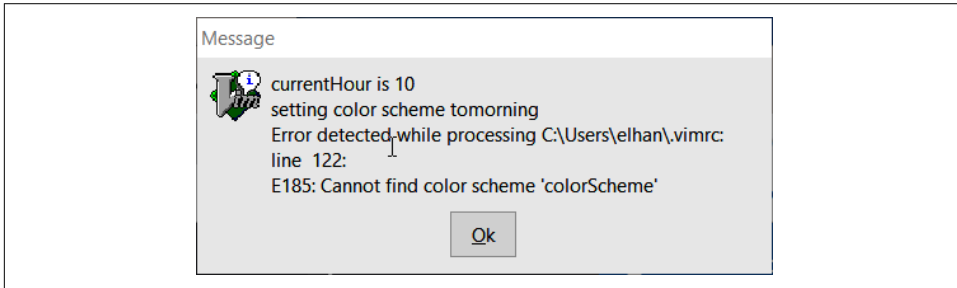
            Ok
```

Figure 12-1. `colorscheme colorScheme` error

We need a way to execute a Vim command that refers to a variable instead of a literal string such as `darkblue`. Vim gives us the `execute` command for this purpose. When passed a command, it evaluates variables and expressions and substitutes their values into the command. We can exploit this feature along with the concatenation shown in the previous section to pass the value of our variable to the `colorscheme` command:

```
execute "colorscheme " . colorScheme
```

The exact syntax used here (particularly the quotation marks) may be confusing. The `execute` command expects variables or expressions, but `colorscheme` is just a plain string, not a variable or an expression. We don't want `execute` to evaluate `colorscheme`; we just want it to accept the name as is. So we turn the name of the command into a literal string by enclosing it in quotation marks. While we're at it, we add a blank space to the end, before the final quotation mark. This is important because we need a space between the command and the value.

Our variable `colorScheme` must be *outside* the quotation marks so that it's evaluated by `execute`. Think of `execute`'s behavior this way:

- Plain words are evaluated as variables or expressions, and `execute` substitutes their values.
- Quotation marks enclosing strings are taken literally; `execute` does not try to evaluate them to return a value.

Using `execute` fixes our error, and Vim now loads the color scheme as expected.

After loading Vim, you can verify that you loaded the proper color scheme. The `colorscheme` command sets its own variable, `colors_name`. In addition to echoing

values of the variables you set in your script, you can manually execute the `echo` command and examine the `colors_name` variable to see whether our script has in fact executed the correct `colorscheme` command based on the time of day:

```
:echo colors_name
```

Defining Functions

So far we've created a script that works nicely for us. Now let's create code we can execute at any time during a session, not just when Vim starts. We will give an example of this soon, but first we need to create a *function* containing the code of our script.

Vim lets you define your own functions with `function…endfunction` statements. Here is a sample skeleton of a user-defined function:

```
function MyFunction(arg1, arg2...)
  line of code
  another line of code
endfunction
```

We can easily turn our script into a function. Notice that we don't need to pass in any arguments, so the parentheses in the function definition are empty:

```
function SetTimeOfDayColors()
  " progressively check higher values... falls out on first "true"
  let currentHour = strftime("%H")
  echo "currentHour is " . currentHour
  if currentHour < 6
    let colorScheme = "darkblue"
  elseif currentHour < 12
    let colorScheme = "morning"
  elseif currentHour < 18
    let colorScheme = "shine"
  else
    let colorScheme = "evening"
  endif
  echo "setting color scheme to " . colorScheme
  execute "colorscheme " . colorScheme
endfunction
```

> Vim user-defined function names must begin with a capital letter.

Now we have a function defined in our *.gvimrc* file. But if we don't call it, the code will never execute. You call a function with Vim's `call` statement. In our example it would look like:

```
call SetTimeOfDayColors()
```

Now we can set our color scheme at any time, anywhere within a Vim session. One option is just to put the previous `call` line in our *.gvimrc*. The results are the same as our earlier example, where we ran the code without using a function. But in the next section, we'll see a neat Vim trick that calls our function repeatedly so that our color scheme gets set regularly throughout our session, thus changing dynamically throughout the day! Of course, this introduces other problems that we must address.

A Nice Vim Piggybacking Trick

In the previous section we defined a Vim function, `SetTimeOfDayColors()`, which we call once to define our color scheme. What if we want to repeatedly check the time of day and change the color scheme accordingly? Obviously the onetime call in *.gvimrc* doesn't accomplish this. To fix this, we introduce a neat Vim trick using the `statusline` option.

Most Vim users take the Vim status line for granted. By default, `statusline` has no value, but you can define it to display virtually any information available to Vim in the status line. And because the status line can display dynamic information, such as the current line and column, Vim recalculates and redisplays the status line any time the edit status changes. Almost any action in Vim triggers a status line redraw. So we'll use this as a trick to call our color scheme function and change the color scheme dynamically. But as we will soon see, this is an imperfect approach.

The `statusline` accepts an expression, evaluates it, and displays it in the status line. This includes functions. We use this feature to call our `SetTimeOfDayColors()` every time the status line is updated, which is often. Because this feature overrides the default status line and we don't want to lose the valuable information we get by default, let's incorporate a wealth of information into the following initial definition of our status line:

```
set statusline=%<%t%h%m%r\ \ %a\ %{strftime(\"%c\")}%=0x%B\
    \\ line:%l,\ \ col:%c%V\ %P
```

> The definition for `statusline` is split across two lines. Vim considers any line with an initial nonblank character of backslash (\) to be a continuation of the previous line, and it ignores all whitespace up to the backslash. So if you use our definition, make sure it is copied and entered exactly. If you can't get it to work, you can revert to starting with an undefined `statusline`.

You can look up the meaning of the various characters preceded by percent signs in the Vim documentation. The definition produces a status line like the following:

```
ch12.asciidoc   Thu 26 Aug 2021 12:39:26 PM EDT    0x3C line:1, col:1 Top
```

Our focus in this chapter is not on what the status line can display but on exploiting the `statusline` option to evaluate a function.

Now we add our `SetTimeOfDayColors()` function to the `statusline`. By using `+=` instead of a plain equals sign, we add something to the end instead of replacing what we defined earlier:

```
set statusline += \ %{SetTimeOfDayColors()}
```

Now our function is part of the status line. Though it doesn't contribute interesting information to the status line, it now checks the time of day and potentially updates our color scheme as the hour of the day progresses. There are two problems with this:

- We now have a Vim script function that inspects the hour of the day each time the Vim status line gets updated. Earlier, we put some effort into eliminating a few calls to `strftime()` for the sake of efficiency, but now we've added so many calls to our session that the number is dizzying.

- When our session happens to evaluate the `statusline` at the proper hour of the day, it does what we want and changes the color scheme. But as we've defined it, it checks the time and resets the color scheme regardless of whether there's a change.

In the next section, we examine more efficient means to our end by using global variables outside of our function.

Tuning a Vim Script with Global Variables

Vim provides scalar variables (numbers and strings) and arrays. Furthermore, you can specify the scope of a variable.

Variable scopes

Vim variables are fairly straightforward, but there are a few things to know and manage before discussing global variables. Specifically, we must manage our variable's *scope*. Vim defines a variable's scope through a convention that depends on the name's *prefix*. The prefixes include:

`a:`
    A function argument.

`b:`
    A variable recognized in a single Vim buffer.

`g:`
    A variable recognized globally—i.e., it can be referenced *anywhere*.

`l:`

> A variable recognized within a function (a local variable).

`s:`

> A variable recognized within a sourced Vim script.

`t:`

> A variable recognized in a single Vim tab.

`v:`

> A Vim variable—one controlled by Vim (these are also global variables).

`w:`

> A variable recognized in a single Vim window.

> If you do not define a variable's scope with a prefix, it defaults to being global (`g:`) when defined outside a function, and to being local (`l:`) when defined within a function.

Global variables

As we discovered with our last modification to our Vim script, we *almost* have the desired behavior. Our function is called every time the Vim status line is updated, but because that happens quite often, it's problematic on several levels.

First, because it's called so often, we might be concerned about the load it creates on the computer's processor. Fortunately, with today's computers this is unlikely to be of much concern, but it's still probably bad form to redefine the color scheme over and over so often. If this were the only issue, we might consider our script complete and not bother tuning it further. However, it is not.

If you've coded along with the examples, you already know the problem. The constant reestablishment of the color scheme while you move around in the edit session creates a noticeable and annoying flicker, because each definition of the color scheme, even if it's the same as the current color scheme, requires Vim to reread the color scheme definition script, reinterpret the text, and reapply all of the color syntax highlight rules. Even computers with extremely high computing power are unlikely to provide enough graphics processing power to render the constant updating flicker-free. We need to fix this.

We can define our color scheme once and then, within a conditional block, determine each time whether the color scheme changes and consequently needs to be defined and drawn. We do this by taking advantage of the global variable set by the `colorscheme` command: `colors_name`. Let's recast our function to take this into consideration:

```
function SetTimeOfDayColors()
  " progressively check higher values... falls out on first "true"
  let currentHour = strftime("%H")
  if currentHour < 6
    let colorScheme = "darkblue"
  elseif currentHour < 12
    let colorScheme = "morning"
  elseif currentHour < 18
    let colorScheme = "shine"
  else
    let colorScheme = "evening"
  endif
  " if our calculated value is different, call the colorscheme command.
  if g:colors_name !~ colorScheme
    echo "setting color scheme to " . colorScheme
    execute "colorscheme " . colorScheme
  endif
endfunction
```

Now we have a dynamic and efficient function. We make one last improvement in the following section.

Arrays

It would be nice if somehow we could just extract our color scheme value without the extended if…then…else block. With Vim arrays, we can improve the script and make it eminently more readable.

You create an array by defining a variable's value as a comma-separated list of values within square brackets. We introduce an array named Favcolorschemes for our function. We could define it within the scope of the function, but to leave open the possibility of accessing the array elsewhere in our session, we'll define it outside of the function as a global array:

```
let g:Favcolorschemes = ["darkblue", "morning", "shine", "evening"]
```

This line should go in your *.gvimrc* file. Now we can reference any value within the array variable g:Favcolorschemes by its subscript, starting with element zero. For example, g:Favcolorschemes[2] is equal to the string "shine".

Taking advantage of Vim's treatment of math functions, where results of integer division are integers (the remainder gets truncated), we can now quickly and easily get our preferred color scheme based on the hour of the day. Let's look at a final version of our function:

```
function SetTimeOfDayColors()
  " currentHour will be 0, 1, 2, or 3
  let g:CurrentHour = strftime("%H") / 6
  if g:colors_name !~ g:Favcolorschemes[g:CurrentHour]
    execute "colorscheme " . g:Favcolorschemes[g:CurrentHour]
    echo "execute " "colorscheme " . g:Favcolorschemes[g:CurrentHour]
    redraw
  endif
endfunction
```

The `echo` … statement prints the information and "announces" the change that just occurred from the script's actions. The `redraw` statement tells Vim to redraw the screen immediately.

Congratulations! You have built a complete Vim script that takes into consideration many of the factors needed to build any useful script you may want.

Dynamic File Type Configuration Through Scripting

Let's look at another nifty script example. Normally, when editing a new file, the only clue Vim gets to determine the file's type and set `filetype` is the file's extension. For example, *.c* means the file is C code. Vim easily determines this and loads the correct behavior to make it easy to edit a C program.

But not all files require an extension. For example, while it's become common convention to create shell scripts with a *.sh* extension, we don't like or abide by this convention, especially having created thousands of scripts before this convention became popular. Vim is actually sufficiently well trained to recognize a shell script without the crutch of a file extension by looking at the text inside the file. However, it can do so only on the second edit, when the file provides some context for determining the type. Vim scripts can fix that!

Autocommands

In our first script example, we relied on Vim's habit of updating the status line constantly and "hid" our function in the status line to set the color scheme by time of day. Our script to dynamically determine the file type relies on a more formal Vim convention, *autocommands*.

Autocommands include any valid Vim commands. Vim uses *events* to execute commands. Here are some events, all of which trigger an associated command when the event happens:

`BufNewFile`
    When Vim begins editing a new file.

`BufReadPre`
    Before Vim moves to a new buffer.

`BufRead`, `BufReadPost`
    When editing a new buffer, but *after* reading the file.

`BufWrite`, `BufWritePre`
    Before writing a buffer to a file.

`FileType`
>   After setting `filetype`.

`VimResized`
>   After a Vim window size has changed.

`WinEnter, WinLeave`
>   Upon entering or leaving a Vim window, respectively.

`CursorMoved, CursorMovedI`
>   Every time the cursor moves in `vi` command mode or in insert mode, respectively.

Altogether there are almost 80 Vim events. For any of these events, you can define an automatic `autocmd` that executes when that event occurs. The `autocmd` format is:

```
autocmd [group] event pattern [nested] command
```

The elements of this format are:

group
>   An optional command group (described later).

event
>   The event that will trigger *command*.

pattern
>   The pattern matching the filename for which *command* should execute.

`nested`
>   If present, allows this autocommand to be nested within others.

command
>   The Vim command, function, or user-defined script to execute when the event occurs.

Our goal is to identify the file type for any new file we open, so we use * for *pattern*.

The next decision is which event to use to trigger our script. Because we want to try to identify our file type as early as possible, two good candidates suggest themselves: `CursorMoved` and `CursorMovedI`.

`CursorMoved` triggers an event when the cursor moves, which seems wasteful, because merely moving the cursor is not likely to provide more information about a file's type. `CursorMovedI`, in contrast, fires when text is input and therefore seems like the best candidate.

We must write a function to do the work each time. Let's call it `CheckFileType()`. We now have enough information to define our `autocmd` command. It looks like this:

```
autocmd CursorMovedI * call CheckFileType()
```

Checking Options

In our `CheckFileType` function, we need to inspect the value of the `filetype` option.
Vim scripts use special variables to extract values from options by prefixing the
option name (`filetype` in our case) with an ampersand (&) character. Hence we use
the variable `&filetype` in our function.

We start with a simple version of `CheckFileType()`:

```
function CheckFileType()
 if &filetype == ""
    filetype detect
 endif
endfunction
```

The Vim command `filetype detect` is a Vim script installed in the *$VIMRUNTIME*
directory. It runs through many criteria and tries to assign a file type to your file.
Normally this occurs once, so if the file is new and `filetype` cannot determine a file
type, the editing session cannot assign syntax formatting.

There is a problem: we call our function each time the cursor moves during input
mode, continually trying to detect the file type. To solve this, we first check to
see whether the file already has a file type, which would mean that our function
succeeded in its previous execution and therefore does not need to do it anymore. We
won't worry about anomalies, such as a mistaken identification or a file that we start
in one programming language and then decide to change to another.

Let's edit a new shell script file and see the results:

```
$ vim ScriptWithoutSuffix
```

Input the following:

```
#! /bin/sh

inputFile="DailyReceipts"
```

By now, Vim has turned on color syntax, as shown in Figure 12-2.

Figure 12-2. File type of new file detected (MS Windows `gvim`, color scheme: morning)

You can tell from the picture that Vim is using gray for the string, but the printed image does not show that `#! /bin/sh` is blue, `inputFile=` is black, and `"Daily Receipts"` is purple. Unfortunately, these aren't the colors for shell syntax highlighting! A quick check of the `filetype` option through the command `:set filetype` displays the message shown in Figure 12-3.

Figure 12-3. `conf` file type detected (MS Windows `gvim`, color scheme: morning)

Vim assigned file type `conf` to our file, which is not what we want. What went wrong?

If you try this example, you will see that Vim assigned the file type immediately when you entered the first character, `#`, at the first `CursorMovedI` event. Configuration files for Unix utilities and daemons typically use the `#` character to start a comment, so Vim's heuristics assume that a `#` at the beginning of the line is the beginning of a comment in a configuration file. We have to teach Vim to be more patient.

Let's change our function to allow for more context. Instead of trying to detect the file type at the first available opportunity, let's allow the user to enter about 20 characters first.

Buffer Variables

We need to introduce a variable into our function to tell Vim to hold off and *not* try to detect the file type until the `CursorMovedI` autocommand calls the function more than 20 times. Our notion of what is a new file, as well as the number of characters we want to enter into that file, is specific to a buffer. In other words, cursor movement in other buffers of the editing session should not count against the check. Therefore, we use a buffer variable and call it `b:countCheck`.

Next, we revise the function to check for at least 20 moves of the cursor in input mode (implying approximately 20 characters entered), along with checking whether a file type has already been assigned:

```
function CheckFileType()
  let b:countCheck += 1

  " Don't start detecting until approximately 20 chars.
  if &filetype == "" && b:countCheck > 20
    filetype detect
  endif
endfunction
```

But now we get the error shown in Figure 12-4.

Figure 12-4. `b:countCheck` *generates an "undefined" error*

That's a familiar error. As before, we had the gall to increment a variable before it was defined. And this time, it's all our fault because our script is responsible for defining `b:countCheck`. We'll handle this subtlety in the next section.

The exists() Function

It's important to know how to manage all of your variables and functions: Vim requires you to define each one so that it already *exists* before any type of evaluation references it.

We can easily resolve our script error by checking for `b:countCheck`'s existence and assigning it a value with the `:let` command shown earlier:

```
function CheckFileType()
  if exists("b:countCheck") == 0
    let b:countCheck = 0
  endif
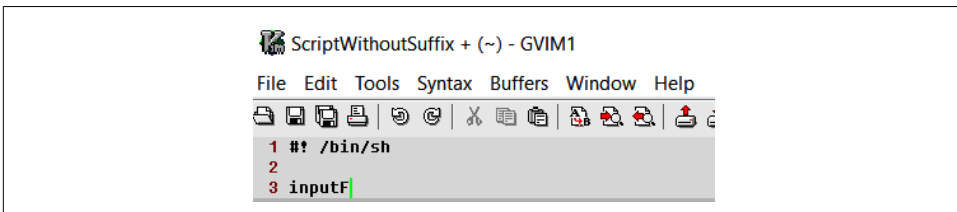
  let b:countCheck += 1

  " Don't start detecting until approx. 20 chars.
  if &filetype == "" && b:countCheck > 20
    filetype detect
  endif
endfunction
```

Now test the code. Figure 12-5 shows the moment before the 20-character limit is reached, and Figure 12-6 shows the effect of entering the 21st character.

Figure 12-5. No file type detected yet (MS Windows `gvim`, *color scheme: morning)*

Figure 12-6. File type detected (MS Windows `gvim`*, color scheme: morning)*

The `/bin/sh` text suddenly has syntax color highlighting. A quick check with `set` `filetype` verifies that Vim has made the correct assignment, as shown in Figure 12-7.

Figure 12-7. Correct detection

For all practical purposes, we have a complete and satisfactory solution, but for good form we add another check to stop Vim from trying to detect a file type after approximately 200 characters have been entered:

```
function CheckFileType()
  if exists("b:countCheck") == 0
    let b:countCheck = 0
  endif
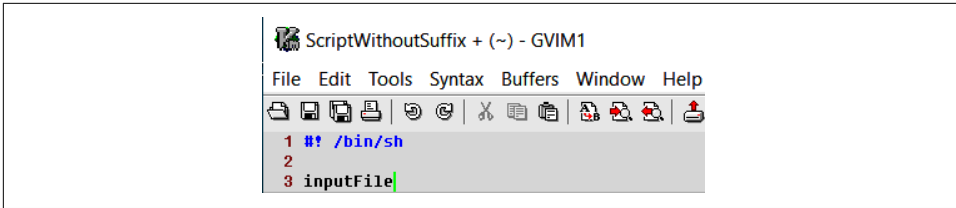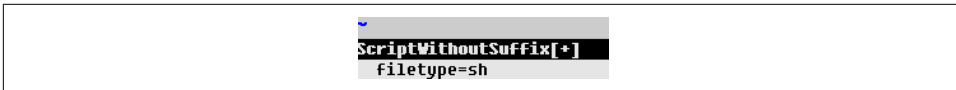
  let b:countCheck += 1

  " Don't start detecting until approx. 20 chars.
  if &filetype == "" && b:countCheck > 20 && b:countCheck < 200
    filetype detect
  endif
endfunction
```

Now, even though `CheckFileType()` is called each time Vim's cursor moves, we incur little overhead because the initial checks exit the function once a file type is detected or the threshold of 200 characters is exceeded. Although this is probably all we need for reasonable functionality and minimal processing overhead, we'll continue to look at more mechanisms to give us a more complete and satisfactory solution that not only incurs minimal overhead but actually "goes away" when we don't need it anymore.

You may have noticed that we've been slightly vague about the exact meaning of our threshold count of 20 characters. This ambiguity is intentional. Because we are counting cursor movements, in input mode it's reasonable to assume each movement of the cursor corresponds to a new character, adding to the "sufficient" context text from which `CheckFileType()` will determine the file type.

However, *all* cursor movement in input mode counts, so any backspacing to correct typing errors also counts against the threshold counter. To confirm this, try our example, type #, and backspace over it and retype it 10 times. The 11th time should reveal a color-coded #, and the file type should now be (incorrectly) set to `conf`.

Autocommands and Groups

Our script so far ignores any side effects introduced by calling our function for every movement of the cursor. We minimized overhead through reasonableness checks that avoid calling the heavy `filetype detect` command unnecessarily. But what if even minimal code for our function is expensive? We need a way to stop calling code when we don't need it. For this we leverage Vim's notion of autocommand *groups* and their ability to remove commands based on their group association.

We modify our example by first associating our function called by the `CursorMovedI` event with a group. Vim provides an `augroup` command to do this. Its syntax is:

```
augroup groupname
```

All subsequent `autocmd` definitions become associated with group *groupname* until the following statement:

```
augroup END
```

There's also a default group for commands not entered within an `augroup` block.

Now we associate our previous `autocmd` command with our own group:

```
augroup newFileDetection
  autocmd CursorMovedI * call CheckFileType()
augroup END
```

Our `CursorMovedI`-triggered function is part of the autocommand group `newFile Detection`. We explore the usefulness of this in the next section.

Deleting Autocommands

To implement our function as cleanly as possible, we want it to remain effective only as long as necessary. We want to undefine its reference once it has exceeded its useful life (that is, as soon as we've either detected a file type or decided that we can't do

so). Vim lets you delete an autocommand simply by referencing the event, the pattern that filenames must match, or its group:

```
autocmd! [group] [event] [pattern]
```

The usual Vim "force" character—an exclamation point (!)—follows the `autocmd` keyword to indicate that commands associated with the group, event, or pattern are to be removed.

Because we previously associated our function with our user-defined group `new FileDetection`, we now have control over it and can remove it by referencing the group in the autocommand remove syntax. We do so with:

```
autocmd! newFileDetection
```

This deletes all autocommands associated with the group `newFileDetection`, which in our case is only our function.

We verify both the definition and removal of our autocommand by querying Vim at startup (when creating the new file) with the command:

```
:autocmd newFileDetection
```

Vim responds as shown in Figure 12-8.

Figure 12-8. Response to `autocmd newFileDetection` command (MS Windows gvim, color scheme: morning)

After a file type has been detected and assigned *or* the threshold of 200 characters has been exceeded, we no longer want the autocommand definition. So we add the final touch to our code. Combining the definition of our `augroup`, our `autocmd` command, and our function, the lines in our *.vimrc* look like:

```
augroup newFileDetection
  autocmd CursorMovedI * call CheckFileType()
augroup END
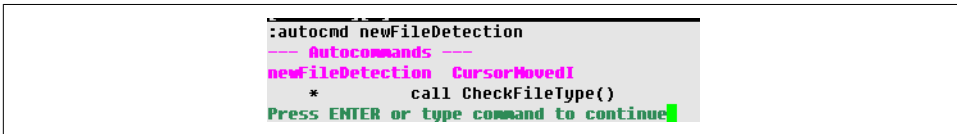
function CheckFileType()
  if exists("b:countCheck") == 0
    let b:countCheck = 0
  endif

  let b:countCheck += 1

  " Don't start detecting until approx. 20 chars.
  if &filetype == "" && b:countCheck > 20 && b:countCheck < 200
    filetype detect
  " If we've exceeded the count threshold (200), OR a filetype has been detected
```

```
    " delete the autocmd!
    elseif b:countCheck >= 200 || &filetype != ""
        autocmd! newFileDetection
    endif
endfunction
```

After the syntax color highlighting begins, we can verify that our function deletes itself by entering the same command as when we entered the buffer:

```
:autocmd newFileDetection
```

Vim's response is shown in Figure 12-9.

Figure 12-9. After the deletion criteria have been met for our `autocommand` group (MS Windows `gvim`, color scheme: morning)

Notice now that no autocommands are defined for the `newFileDetection` group. You can delete the auto group as follows:

```
augroup! groupname
```

but doing so does *not* delete the associated autocommands, and Vim will create an error condition each time those autocommands are referenced. Therefore, make sure to delete the autocommands within a group before deleting the group.

 Do not confuse deleting autocommands with deleting auto groups.

Congratulations! You have completed your second Vim script. This script extends your Vim knowledge and gives you a peek at the different features accessible with scripting.

Some Additional Thoughts About Vim Scripting

We've covered only a small corner of the entire Vim scripting universe, but we hope you get the sense of Vim's power. Virtually everything you can do interactively using Vim can be coded in a script.

In this section we look at a nice example included in the built-in Vim documentation, discuss in more detail concepts we touched on earlier, and look at a few new features.

A Useful Vim Script Example

Vim's built-in documentation includes a handy script we think you'll want to use. It specifically addresses keeping a current timestamp in the `<meta>` line of an HTML file, but it could easily be used for many other types of files for which it is useful to have the most recent modification time of the file within the text of that file.

Here is the example essentially intact (we have modified it slightly):

```
autocmd BufWritePre,FileWritePre *.html    mark s|call LastMod()|'s
fun LastMod()
  " if there are more than 20 lines, set our max to 20, otherwise, scan
  " entire file.
  if line("$") > 20
    let lastModifiedline = 20
  else
    let lastModifiedline = line("$")
  endif
  exe "1," . lastModifiedline . "g/Last modified: .*/s//Last modified: " .
  \ strftime("%Y %b %d")
endfun
```

Here's a brief breakdown of the `autocmd` command:

`BufWritePre`, `FileWritePre`
> These are the events for which the command is triggered. In this case, Vim executes the autocommand *before* the file or buffer gets written to the storage device.

`*.html`
> Execute this autocommand for any file whose name ends in *.html*.

`mark s`
> We changed this for readability from the original. Instead of `ks`, we used the equivalent but more obvious command `mark s`. This simply creates a marked position named `s` in the file so we can return to this point later.

`|`
> Pipe characters separate multiple Vim commands that are executed within an autocommand definition. These are simple separators with no relationship to Unix shell pipes.

`call LastMod()`
> This calls our user-defined `LastMod()` function.

`|`
> Same as previous.

`'s`
> Return to the line we marked with the name `s`.

It's worth verifying this script by editing a *.html* file, adding the line:

```
Last modified:□
```

and issuing the `w` command.

> This example is useful, but it's not canonically correct in relation to its stated goal of substituting the HTML `<meta>` statement. More appropriately, if it were indeed meant to address a `<meta>` statement, the substitution should look for the `content=…` part of the `<meta>` statement. Still, this example is a good start toward solving that problem and is a useful example for other file types.

More About Variables

We now discuss in more detail what makes up Vim variables and how they're used. Vim has five variable types:

Number
    A signed 32-bit number. This number can be represented in decimal, hexadecimal (e.g., `0xffff`), or octal (e.g., `0177`). If supported by the compiler, Vim supports 64-bit numbers. The ex command `:version` shows whether 64-bit is supported or compiled into the editor. See Figure 12-10.


```
break           +netbeans_intg      +sy
indent          +num64              +ta
cmds            +packages           +ta
```

Figure 12-10. Results from `:version` showing 64-bit number support (WSL Ubuntu Linux, color scheme: zellner)

String
    A string of characters.

Funcref
    A reference to a function.

List
    This is Vim's version of an array. It is an ordered list of values and can contain any mix of Vim values as elements.

Dictionary
    This is Vim's version of a *hash*, often also referred to as an *associative array*. It is an unordered collection of value pairs, the first being a *key* that can be used to retrieve an associated *value*.

Expressions

Vim evaluates expressions in a fairly straightforward way. An expression can be as simple as a number or literal string, or it can be as complex as a compound statement, itself composed of expressions.

It is important to note that Vim's math functions work with integers only. If you want floating-point and precision, you need to use extensions, such as system calls to external math-capable routines.

Extensions

Vim offers a number of extensions and interfaces to other scripting languages. Notably, these include Perl, Python, and Ruby, three of the most popular scripting languages. See Vim's built-in documentation for details on usage.

A Few More Comments About autocmd

In the section "Dynamic File Type Configuration Through Scripting" on page 300, we used Vim's `autocmd` command to key on events from which our user-defined functions are called. This is very powerful, but do not discount simpler uses of `autocmd`. For example, you can use `autocmd` to tune specific Vim options for different file types.

A good example might be to change the `shiftwidth` option for different file types. File types with copious indentation and nesting levels may benefit from more modest indentation. You may want to define your `shiftwidth` as 2 for HTML to prevent code from "walking" off the right side of the screen, but use a `shiftwidth` of 4 for C programs. To accomplish this distinction, include these lines in your *.vimrc* or *.gvimrc* file:

```
autocmd BufRead,BufNewFile *.html set shiftwidth=2
autocmd BufRead,BufNewFile *.c,*.h set shiftwidth=4
```

Internal Functions

In addition to all the Vim commands, you have access to about 200 built-in functions. It is beyond our scope to identify and document all of these functions, but knowing what categories or types of functions are available is useful. The following categories are taken from the Vim built-in help file, *usr_41.txt*:

String manipulation
> All of the standard string functions that programmers expect are included in these functions, from conversion routines to substring routines and more.

List functions

   This is an entire array of array functions. They mirror closely the array functions found in Perl.

Dictionary (associative array) functions

   These functions include extraction, manipulation, verification, and other types of functions. Again, these closely resemble Perl's hash functions.

Variable functions

   These functions are "getters" and "setters" to move variables around in Vim windows and buffers. There is also a `type()` function to determine variable types.

Cursor and position functions

   These functions allow moving around in files and buffers, and creating marks so that positions can be remembered and returned to. There are also functions that give positional information (e.g., cursor line and column).

Text in current buffer functions

   These functions manipulate text within buffers—for example, changing a line, retrieving a line, and so on. There are also search functions.

System and file manipulation functions

   These include functions to navigate the operating system on which Vim is running, such as finding files within paths, determining the current working directory, and creating and deleting files. This group includes the `system()` function, which passes commands to the operating system for external execution.

Date and time functions

   These do a wide variety of manipulations of date and time formats.

Buffer, window, and argument list functions

   These functions provide mechanisms to gather information about buffers, and the arguments for each one. For example, when Vim starts, the list of files comprises its argument list, and the function `argc()` returns the count of that list. Argument list functions are specific to arguments passed from the Vim command line. The buffer functions provide information specific to buffers and windows. There are 25 functions. For more detailed information, search for *Buffers, windows, and the argument list* in the Vim's *usr_41.txt* help file. Or you can quickly get to the types of Vim functions with the `ex` command:

```
:help function-list
```

Command-line functions

   These functions get command-line position, the command line, and the command-line type and set the cursor position within the command line.

QuickFix and location lists functions
    These functions retrieve and modify the QuickFix lists.

Insert mode completion functions
    These functions are used for command and insertion completion features.

Folding functions
    These functions give information for folds and expand text displayed for closed folds.

Syntax and highlighting functions
    These functions retrieve information about syntax highlighting groups and syntax IDs.

Spelling functions
    These functions find suspected misspelled words and offer suggested correct spellings.

History functions
    These functions get, add, and delete history items.

Interactive functions
    These functions provide an interface to the user for activities such as file selection.

GUI functions
    There are three simple functions here to get the name of the current font, get the GUI window *x* coordinate, and get the GUI window *y* coordinate.

Vim server functions
    These functions communicate with a (possibly) remote Vim server.

Window size and position functions
    These functions get window information and allow for saving and restoring window "views."

Various functions
    These are the miscellaneous "other" functions that don't fit nicely in the previous categories. They include functions such as `exists()`, which checks for the existence of a Vim item, and `has()`, which checks to see whether Vim supports a certain feature.

Resources

We hope we've piqued enough interest and provided enough information to get you started with Vim scripts. An entire book could be devoted to the subject of Vim scripting. Luckily, there are other resources to turn to for help.

A good starting point is to go to the source of Vim itself and visit the pages specifically dedicated to scripting. Here you will find over two thousand scripts available for download. The entire body of work is searchable, and you are invited to participate by rating scripts and even contributing your own.

We also remind you that the built-in Vim help is invaluable. The most productive help topics we recommend are:

```
help autocmd
help scripts
help variables
help functions
help usr_41.txt
```

And don't forget the myriad Vim scripts in the Vim runtime directories. All of the files with the suffix .vim are scripts, and these provide an excellent and fertile playground for learning to code by example.

Go play. It's the best way to learn.

CHAPTER 13

Other Cool Stuff in Vim

Chapters 8 through 12 covered powerful Vim features and techniques we think you should know to make effective use of the editor. From real-time spellchecking (with suggested corrections) to editing binary files and managing the state of your Vim session, this chapter takes a lighter look at Vim. It's a catchall for some of the features that didn't fit into previous topics, ideas about editing and the Vim philosophy, and some fun things about Vim (not that the earlier chapters weren't fun!).

Spell It! (i-t)

Vim's spellchecking excels in speed and flexibility. Per Vim's ownspellchecker built-in help, Vim recommends replacing the `vimspell` plug-in with Vim's built-in spell-checking features. See the help file *spell.txt*, or find help with the `ex` command:

    :help **spell**

Vim defaults to no spellchecking. Turn spellchecking on with the `ex` command:

    :**setlocal spell**

and the spellchecking region:

    :**setlocal** spelllang=en_us

Vim flags "bad" words (because one person's misspelled word is another one's "good" word, hence a distinction between "bad" and "good" rather than "correct" and "incorrect"), uncapitalized words starting sentences, "rare" words (don't ask), and "local" (regional) words. See Figure 13-1 for an example of how Vim highlights "bad" and "cap" flagged words, and see Figure 13-2 for an example of highlighted words.

Figure 13-1. Vim spellchecking syntax highlighting (MacVim, color scheme: zellner)

Figure 13-2. Highlighted "bad" words

Once spellchecking is turned on, you advance to the next or previous bad word with the `vi` command mode commands `]s` and `[s`, respectively. When the cursor is positioned anywhere in a bad word, the `vi` command `z=` suggests words in a numbered list to replace the bad word. Type the corresponding number and hit ENTER to replace the bad word. Or if you are in a GUI Vim session or have the mouse enabled, select the replacement word by clicking on your choice. To cancel the operation, type ESC or ENTER with no selection.

Vim manages spelling lists by loading encoded word files into memory and applying two main algorithms to detect misspellings. One is fast; the other is slow. Vim lets you turn either one on or off. The *fast* spellchecking assumes that misspelled words will closely match the correctly spelled word and that the error is likely a transposition of characters or a missing character. These misspelled words are considered to be a "short distance" from correctness, and hence the algorithm is efficient and fast.

The *slow* algorithm assumes that the word could be a "large" distance from correctness. For example, you may have spelled the word phonetically, without knowing what is really correct.

Per Vim's documentation, if you are a good speller, your errors are likely to be of the fast-algorithm kind, and Vim recommends for efficiency that you use only that option. You select your preferred spellchecking method by choosing one of `fast` or `double` as the value of the Vim option `spellsuggest`.[1]

---

1 There is a third method, `best`, which Vim suggests is best for English text, but the fastest still is `fast`.

We have listed the more common vi command-mode commands in Table 13-1.

Table 13-1. Common Vim command-mode spellchecking commands

Command	Action
]s	Advance the cursor to the next occurrence of a misspelled word.
[s	Advance the cursor to the previous occurrence of a misspelled word.
zg	Add the word under the cursor to the list of good words.
zG	Add the word under the cursor to the list of good words in the `internal-wordlist` (see the Vim help). Words added to the `internal-wordlist` are considered transient and are discarded upon exiting Vim.
zw	Add the word under the cursor to the list of bad words. If this word is in the good words list, it will be removed from there.
zW	Add the word under the cursor to the list of bad words in the `internal-wordlist`. As with zG, this addition is discarded upon exiting Vim.
[*number*] z=	Display the list of suggestions for replacement of a bad word. Vim displays the suggestions as a numbered list, and you select the replacement by entering the corresponding number. If you precede z= with a number, Vim automatically replaces the bad word with the suggested word matching *number*.

Some notes:

- The Vim option `wrapscan` applies to]s and [s. That is, if there are no more misspelled words between the current location and the end of the buffer (or the beginning, depending on the direction of the command), the cursor is not advanced.

- Vim distinguishes words as "good" or "bad," rather than "correctly spelled" or "incorrectly spelled," because there may be terms that are correct in context but are not necessarily actual words.

- For zg and zG, the words you add to both the good word list and the bad word list are added to the file defined in the Vim option `spellfile`. This keeps these added words separate from the more global Vim spelling files.

- For z=, if you are using GUI Vim or have enabled mouse actions, you can select the replacement by clicking on the suggested word.

  Vim documentation hints that the most common numeric prefix might be 1, assuming that the first suggestion is likely the one that would replace the bad word.

Table 13-2 lists the ex commands and how they are used.

Table 13-2. Common Vim ex commands for spellchecking

Command	Action
`:[`*n*`]spellgood` *word*	Add *word* to the good word list. If the command is preceded by a count, add *word* to the *n*th file in the list of files defined by `spellfile`. If there isn't a corresponding *n*th file (e.g., a count of three, but only two files are defined by the Vim `spellfile` option), Vim flags an error, and no word is added.
`:spellgood!` *word*	Add *word* to the good word list in `internal-list`. Vim discards the `internal-list` after each session.
`:spellwrong` *word*	Add *word* to the bad word list.
`:spellwrong!` *word*	Add *word* to the bad word list in `internal-list`. Vim discards the `internal-list` after each session.

Vim documentation gives detailed instructions beyond simply checking for misspelled words. For example, you can define your own word files, and Vim's help file *spell.txt* in section 3, "Generating a spell file," shows how to create word files from starter sets like the words from OpenOffice, for example. You can use freely available files, create your own, or go with some combination of the two. Visit Vim's documentation (`:help spell`) for more detailed descriptions of setting up custom spelling configurations and of more available commands and options.

For a Different Take on Words, Try Thesaurus

Not to be confused with spellchecking, Vim also provides word completion by thesaurus. For more detailed discussion on Vim's thesaurus options, see "Completion by thesaurus" on page 263. It's interesting, alluring, amusing, attractive, beautiful, compelling, curious, delightful, engaging, exotic, fascinating, impressive, intriguing, lovely, pleasing, provocative, readable, refreshing, stimulating, and striking. ☺

Editing Binary Files

Officially, Vim, like `vi`, is a *text* editor. But in a pinch, Vim also lets you edit files containing data that is normally unreadable by humans.

Why would you ever want to edit a binary file? Aren't binary files binary for a reason? Aren't binary files typically generated by some application in a well-defined and specific format?

It's true that binary files are typically created by a computerized or analog process and are not intended to be edited manually. For example, digital cameras often store pictures in JPEG format, a compressed binary format for digital pictures. These are binary, but they have well-defined sections or blocks in which standard information is stored (that is, they do if they're implemented according to specification). Digital pictures in JPEG format store picture meta-information (time of picture, resolution, camera settings, date, etc.) in reserved blocks separate from the compressed digital

picture data. A practical application might use Vim's binary file editing feature to edit a directory of JPEG pictures to change all of the `year` fields in the "created" block to correct the picture's "date of creation" field.

While we enjoy Vim's binary editing feature, we do not present an in-depth discussion of potential serious issues to consider while editing binary files. For example, some binary files contain digital signatures or checksums to ensure file integrity. Editing the files risks damaging their integrity and rendering them unusable. Therefore, do not consider this an endorsement of casual binary editing.

Figure 13-3 shows an editing session on a JPEG file. Notice how the cursor is positioned over the date field. You can directly edit information about this picture by changing the fields shown here.

Figure 13-3. Editing a binary JPEG file

For power users familiar with a particular binary format, Vim can be extremely handy for making changes directly that might otherwise require tedious, repetitive access with other tools.

Binary Editing to the Rescue

One of us had a real-life experience in which Vim's binary editing feature saved the day. He was tasked with porting a legacy application from a deprecated computer to a new computer. The application was partially composed of many Python classes (compiled *.pyc* files). In a classic IT dilemma, he discovered that no original Python code existed, and hence there was no option to port the classes by compiling them on the new computer.

The classes actually were executable on the new computer but had old, obsolete computer names and addresses embedded within them. On a hunch, your author edited the classes in binary mode and discovered that all the old host names had the same string length as the new computer names. After a simple mass substitution and save, the Python classes worked perfectly on the new system. Yes, it was a stroke of luck that the computer names, old and new, were the same length. Still, without Vim, it would have been a much more difficult task.

There are two main ways to edit binary files. You can set the `binary` option from the Vim command line:

```
:set binary
```

or start Vim with the `-b` option.

To facilitate binary editing *and* protect Vim from damaging the files' integrity, Vim sets the following options accordingly:

- The `textwidth` and `wrapmargin` options are set to zero. This stops Vim from inserting spurious newline sequences into the file.
- The `modeline` and `expandtab` options are unset (`nomodeline` and `noexpandtab`). This stops Vim from expanding tabs to `shiftwidth` spaces and prevents it from interpreting commands in a modeline, which potentially would set options that introduce unexpected and unwanted side effects.

> Be careful when moving from window to window, or from buffer to buffer, while using binary mode. Vim uses entry and exit events to set and change options for switching buffers and windows, and you may confuse it into removing some of the protections just listed. We recommend a single-window, single-buffer session when editing binary files.

Digraphs: Non-ASCII Characters

Do you say that *Messiah* is composed by George Frideric *Händel*, not George Frideric *Handel*? Do you think your *résumé* conveys a little more cachet than a *resume*? Use Vim's digraphs to enter special characters.

Even English-language text files occasionally need a special character, especially when making references to a globalized world. Text files in languages other than English need scads of special characters.

The term *digraph* traditionally describes a two-letter combination that represents a single phonetic sound, such as the *ph* in "digraph" or "phonetic." Vim borrows the notion of two-letter combinations to describe its input mechanism for characters with special characteristics, typically accents or other markings such as the umlaut on *ä*. These special marks are properly called *diacritics*, or *diacritical marks*. In other words, Vim uses digraphs to create diacritics. (Glad we could clear that up.)

Vim lets you enter special characters (diacritics) in a number of ways, and two of them are relatively straightforward and intuitive. One defines a digraph using a prefix character (CTRL-K). The second uses the BACKSPACE key between two keyboard characters. (The other methods are more suited to entering characters by their raw

numerical values, specified as decimal, hexadecimal, or octal numbers. While powerful, these methods do not lend themselves to easy mnemonics for digraphs.)

The first input method for diacritics is a three-character sequence consisting of CTRL-K, the base letter, and a punctuation character indicating the accent or mark to be added. For example, to create a *c* with a cedilla (ç), enter CTRL-K C ,. To create an *a* with a grave accent (à), enter CTRL-K a !.

Greek letters are created by entering a corresponding Latin letter followed by an asterisk (for instance, enter CTRL-K P * for a lowercase π). Russian letters are created by entering a corresponding Latin letter followed by an equals sign (=) or, in a few places, a percent sign (%). Use CTRL-K ? SHIFT-I to enter an inverted question mark (¿) and CTRL-K S S to enter a German sharp *S* (ß).

To use Vim's second method, set the `digraph` option:

```
:set digraph
```

Now you create special characters by typing the first character of the two-character combination, then a backspace character (BACKSPACE), and then the punctuation that creates a mark. Thus, enter *ç* using C BACKSPACE , and *à* through A BACKSPACE !.

Setting the `digraph` option doesn't preclude you from entering digraphs with the CTRL-K method. Consider using *only* the CTRL-K method if your typing is less than stellar. Otherwise, you may find yourself inadvertently entering digraphs more often than you want as you backspace and type corrections.

Use the `:digraph` command to show all the default sequences; more verbose descriptions can be obtained with `:help digraph-table`. Figure 13-4 shows a partial list from the `:digraph` command.

In the display, each digraph is represented by three columns. The display is a bit jumbled because Vim jams as many three-column combinations on each line as the screen permits. For each of the groups, column one shows the digraph's two-character combination, column two displays the digraph, and column three lists the decimal Unicode value for the digraph.

```
NU ^@  10     SH ^A   1     SX ^B    2
LF ^@  10     VT ^K  11     FF ^L   12
D4 ^T  20     NK ^U  21     SY ^V   22
RS ^^  30     US ^_  31     SP      32
'!  `  96     (! {  123     !! |   124
IN <84> 132   NL <85> 133   SA <86> 134
S2 <8e> 142   S3 <8f> 143   DC <90> 144
SS <98> 152   GC <99> 153   SC <9a> 154
~!  ¡  161    Ct ¢  162     c| ¢  162
|| ¦  166     SE §  167     ': ¨  168
Rg ®  174     'm ¯  175     -= ¯  175
'' ´  180     My µ  181     PI ¶  182
>> »  187     14 ¼  188     12 ½  189
```

Figure 13-4. Vim digraphs (MacVim, color scheme: zellner)

For your convenience, Table 13-3 lists the punctuation to use as the final character in the sequence to enter the most commonly needed accents and marks.

Table 13-3. How to enter accents and other marks

Mark	Example	Character to enter as part of the digraph
Acute accent	fiancé	Apostrophe (')
Breve	publică	Left parenthesis (
Caron	Dubček	Less-than sign (<)
Cedilla	français	Comma (,)
Circumflex or caret	português	Greater-than sign (>)
Grave accent	voilà	Exclamation point (!)
Macron	ātmā	Hyphen (-)
Stroke	Søren	Slash (/)
Tilde	señor	Question mark (?)
Umlaut or diaeresis	Noël	Colon (:)

Editing Files in Other Places

Thanks to seamless integration of network protocols, Vim lets you edit files on remote machines just as if they were local! If you simply specify a URL for a filename, Vim opens it in a window and writes your changes to it on the remote system. (This depends on your access rights.) For instance, the following command edits a shell script owned by user `elhannah` on the system `flavoritlz`. The remote machine offers the SSH secure protocol on port 122 (this is a nonstandard port, providing additional security through obscurity):

```
$ vim scp://elhannah@flavoritlz:122//home/elhannah/bin/scripts/manageVideos.sh
```

Because we're editing a file in elhannah's home directory on the remote machine, we can shorten the URL by using a simple filename. It's treated as a pathname relative to the user's home directory on the remote system:

```
$ vim scp://elhannah@flavoritlz:122/bin/scripts/manageVideos.sh
```

Let's dissect the full URL so you can learn how to build URLs for your particular environment:

scp:

The first part, up to the colon, represents the transport protocol. In this example, the protocol is scp, a file copy protocol built on the Secure Shell (SSH) protocol. The following : is required.

//

This introduces host information, which for most transport protocols takes the form [*user*@]*hostname*[:*port*].

elhannah@

This is optional. For secure protocols such as scp, it specifies which user to log in as on the remote machine. When omitted, it defaults to your username on the local machine. When you are prompted for a password, you must enter the user's password on the remote machine.

flavoritlz

This is the remote machine's symbolic name, and it can also be specified as a numeric IP address, e.g., 192.168.1.106.

:122

This is optional and specifies the port on which the protocol is provided. The colon separates the port number from the preceding hostname. All standard protocols use well-known ports, so this element of the URL can be omitted if the standard port is used. In this example, 122 is *not* the standard port for the scp protocol, and because the administrator of the flavoritlz system has chosen to provide the service on port 122, this specification is required.

//home/elhannah/bin/scripts/manageVideos.sh

This is the file on the remote machine we want to edit. We start with two slashes because we're specifying an absolute path. A relative path or simple filename requires only a single slash to separate it from the preceding hostname. A relative path is relative to the home directory of the user that you logged in as. (In this example, it would be relative to */home/elhannah*.)

Here is a partial list of the supported protocols:

`ftp:` *and* `sftp:`
> Regular FTP and secure FTP.

`scp:`
> Secure remote copy over SSH.

`http:`
> File transfer using standard browser protocol.

`dav:`
> A relatively new but popular proposed open standard for web transfer.

`rcp:`
> Remote copy. Note that this protocol is insecure; *you should never use it.*

What we've described so far is enough to allow remote editing, but the process may not be as transparent as editing a file locally. That is, because of the intervening requirement to move data from remote hosts, you may be prompted for passwords to do your work. This can become tedious if you are used to periodically writing your file to disk while editing, as each of the "writes" is interrupted to prompt you to enter a password to complete the transaction.

All of the transport protocols in the preceding list allow you to configure the service to allow password-free access, but the details vary. Use the service's documentation for specific protocol details and configurations.[2]

Navigating and Changing Directories

If you've used Vim a lot, you may have accidentally discovered that you can view a directory and move through it using keystrokes similar to those used with files.

Let's consider a directory containing two repository directories, */home/elhannah/.git/vim* (these are two different *git* directories). Edit */home/elhannah/.git/vim* with:

```
$ vim /home/elhannah/.git/vim
```

Figure 13-5 is a partial screenshot of something similar to what you might see.

---

2 We have verified and successfully set up `scp:` remote access to edit remote files without needing to enter passwords. Setting up `scp:` (as well as the other protocols) remains outside our scope, but we consider it worthwhile for convenient and transparent remote editing.

Figure 13-5. Vim "editing" the "vim" directory (WSL Ubuntu Linux, color scheme: zellner)

Vim displays three types of information: introductory comments (surrounded by equals signs), directories (displayed with trailing slashes), and files. Each directory or file is on its own line.

There are many ways to use this feature, but with little effort you can be immediately and intuitively productive with the standard Vim motion commands (e.g., `w` to move to the next word, `j` or the down arrow to move down one line) and by clicking the mouse over entries. Here are some particular features of directory mode:

- When the cursor is positioned over a directory name, move to that directory by pressing the ENTER key.

- If the cursor is over a filename, pressing ENTER edits that file.

> If you want to keep the directory window around for further work in that directory, edit the file under the cursor by typing `o`, and Vim will split the window, editing the file in the newly created window. This is also true for moving to another directory when the cursor is over a directory name; Vim splits the window and "edits" the directory to which you moved in the new window.

- You can delete and rename files and directories. You do so typing SHIFT-R. Probably a little counterintuitively, Vim creates a command-line prompt with which you perform the rename. It should look something like Figure 13-6.

  To complete the rename, edit the second command-line argument.

  Deleting a file works similarly. Simply position the cursor over the filename you want to delete and type SHIFT-D. Vim prompts you with a verification dialog to delete the file. As with the rename function, Vim prompts for verification in the command-line area of the screen.

```
appa.asciidoc
[learning-the-vi-and-vim-editors-8e[-][RO]    Tue Jun 29 10:41:25 2021           0x78  line:10,   col:1 Top type:[netrw]
Moving /home/elhannah/.git/vim/oreilly/learning-the-vi-and-vim-editors-8e/xyzzy.txt to : /home/elhannah/.git/vim/oreilly
/learning-the-vi-and-vim-editors-8e/xyzzy.txt
```

Figure 13-6. Prompt for rename in "edit" directory (WSL Ubuntu Linux, color scheme: zellner)

- One advantage of editing directories is quick access to files with Vim's search function. For example, suppose you want to edit the file *ch12.asciidoc* in the */home/elhannah/.git/vim/oreilly/learning-the-vi-and-vim-editors-8e* directory described earlier. To quickly navigate to and edit this file, you can search for part or all of the filename. In this case we search simply for the number 12:

      /12

  and with the cursor over that filename, press ENTER or O.

> When you read the online help for directory editing, you will see that Vim describes it as part of the entire suite of editing files with network protocols, which was described in the previous section. We have made directory editing its own topic because it is useful, and because it could get lost in the large volume of detail about network protocol editing.

Backups with Vim

Vim helps protect you from unintentionally losing data by letting you make a backup of the files you edit. For an editing session that has gone terribly wrong, this can be useful because you can recover your previous file.

Backups are controlled by the settings of two options: `backup` and `writebackup`. Where and how backups are created is controlled by four other options: `backupskip`, `backupcopy`, `backupdir`, and `backupext`.

If both the `backup` and `writebackup` options are off (i.e., `nobackup` and `nowrite backup`), Vim makes no backup files. If `backup` is on, Vim deletes any old backups and creates a backup for the current file. If `backup` is off and `writebackup` is on, Vim creates a backup file for the duration of the editing session and deletes the backup afterward.

The `backupdir` is a comma-separated list of directories in which Vim creates backup files. For example, if you want backups to always be created in your system's temporary directory, set `backupdir` to `C:\TEMP` for Windows or to `/tmp` for Unix and GNU/Linux.

If you'd like to always create a backup of your file in the current directory, you can specify "." (a dot) as your backup directory. Or you could try to create a backup in a hidden subdirectory first if it exists, and then in the current directory if the hidden subdirectory doesn't exist. Do this by defining `backupdir`'s value to be something like `./.mybackups,.` (the single dot at the end denotes the file's current directory). This is a flexible option that supports many strategies for defining backup locations.

If you want to make backups while editing but not for all files, use the `backupskip` option to define a comma-separated list of patterns. Vim will not make a backup of any file matching any of the patterns. For example, you may never want to back up any files edited in the */tmp* or */var/tmp* directories. Prevent Vim from doing so by setting `backupskip` to `/tmp/*,/var/tmp/*`.

By default, Vim creates your backup with the same filename as the original and the suffix ~ (a tilde). This is a fairly safe suffix, because filenames ending in that character are rare. Change the suffix to your preference with the `backupext` option. For example, if you want your backups to have the suffix *.bu*, set `backupext` to the string `.bu`.

Finally, the `backupcopy` option defines *how* a backup copy is created. We recommend setting this option to `auto` to let Vim make a calculated choice of the best method for the backup.

HTML Your Text

Have you ever needed to present your code or text to a group? Have you ever tried to do a code review but were using someone else's Vim configuration and couldn't figure it out? Consider converting your text or code to HTML and viewing it from a browser.

Vim provides three methods to create an HTML version of your text. They all create a new buffer with the same name as the original file and the added suffix *.html*. Vim splits the current session window and displays the HTML version of the file in the new window:

`gvim` *"Convert to HTML"*
> This is the friendliest method and is built into the `gvim` graphical editor (described in Chapter 9). Open the Syntax menu in `gvim` and select "Convert to HTML."

`2html.vim` *script*
> This is the underlying script invoked by the "Convert to HTML" menu option described in the previous item. Invoke it through the command:

```
:runtime!syntax/2html.vim
```

It doesn't accept a range; it converts the whole buffer.

tohtml *command*

This is more flexible than the 2html.vim script, because you can specify an exact range of lines you want to convert. For instance, to convert lines 25 through 44 of a buffer, enter:

```
:25,44tohtml
```

> While the Vim distribution still includes tohtml.vim in the plug-in (and autoload) directories, we were not able to successfully use this feature. Your mileage may vary. The other conversions do work.

One advantage of using gvim for HTML conversion is that the GUI lets it accurately detect colors and create correct corresponding HTML directives. These methods still work in a non-GUI context, but the results are less assured to be accurate and may not be very useful.

> It's up to you to manage the newly created file. Vim does not save it for you; it merely creates a buffer. We recommend providing a management policy to save and synchronize HTML versions of your text files. For example, you could create some autocommands to trigger the creation and saving of your HTML files.

The saved HTML file can be viewed in any web browser. Some people may not be familiar with ways to open files on the local system in their browsers. It's quite easy, though: virtually all browsers offer an Open File menu option in the File menu and display a file selection dialog to let you navigate to the folder containing the HTML file. If you plan on using this feature on a regular basis, we recommend building up a collection of bookmarks for all of your files.

What's the Difference?

Changes between different versions of a file are often subtle, and a tool that lets you view precise differences at a glance could save hours of work. Vim integrates the well-known Unix diff command into a very sophisticated visualization interface invoked through its vimdiff command.

There are two equivalent ways to invoke this feature—as a standalone command and as an option to Vim:

```
$ vimdiff old_file new_file
$ vim -d old_file new_file
```

Typically, the first file to be compared is an old version of a file, and the second is a newer version, but that is by convention only. Indeed, it's possible to make a case for reversing the order.

Figure 13-7 shows an example of `vimdiff` output. Because of limited real estate, we've squeezed the width and turned off Vim's `wrap` option to allow illustration of the differences.

Figure 13-7. `vimdiff` results (WSL Ubuntu `gvimdiff`, color scheme: zellner)

Although the figure does not convey the full impact of the visual content in the printed book, it shows some key characteristic behaviors:

- On line 4, you can see the word *new* on the left line that isn't in the right line. This is a highlighted (in red) word indicating the difference between the two lines. Similarly, on line 32, the righthand line highlights in red the word *reflect* that is not in the left line.

- On line 11 of both sides, Vim has created a 15-line fold. These 15 lines in both files are identical, so Vim folds them to maximize useful "diff" information on the screen.

- Lines 41–42 on the left are highlighted, whereas in the corresponding positions on the right, strings of hyphens (-) indicate that the lines are missing. The line numbering differs from this point on, because the right side has two lines fewer, but corresponding lines in the two files still line up horizontally.

- Line 49 on the left side, corresponding to line 47 on the right, shows another fold, this one 2010 lines, which shows that for the remaining 2010 lines of both files the content is identical.

The `vimdiff` feature comes with all Unix-like Vim installations because the `diff` command is a Unix standard. Non-Unix Vim installations should come with Vim's own version of `diff`. Vim allows drop-in replacements of `diff` commands as long as they create standard `diff` output.

The `diffexpr` variable defines the replacement expression for the default `vimdiff` behavior and is typically implemented as a script that operates on the following variables:

`v:fname_in`
> The first input file to be compared.

`v:fname_new`
> The second file to be compared.

`v:fname_out`
> A file that captures the `diff` output.

viminfo: Now, Where Was I?

Most text editors start editing files at line 1, column 1. That is, each time the editor starts, the file is loaded and editing begins from line 1. If you edit a file many times, progressing through it, you would find it more convenient to begin your new session where the last one ended. Vim lets you do just that.

There are two different methods to save session information for future use: the `viminfo` option and the `:mksession` command. We look at both in this section.

The viminfo Option

Vim uses the `viminfo` option to define what, how, and where to save editing session information. The option is a string with comma-delimited parameters that tell Vim how much information to save and where to save it. Some of `viminfo`'s suboptions are defined by the following:

<n

> Tells Vim to save lines for each register, up to a maximum of *n* lines.

> If you do not specify any value for this option, *all* lines are saved. While at first this may seem a normal and reasonable default, consider whether you commonly edit very large files and make large changes to those files. For example, if you commonly edit a 10,000-line file and delete all lines (possibly to pare it down from rapid growth caused by some external application) and then save it, all 10,000 lines get saved in the *.viminfo* file for that entry. If you do this often for many files, the *.viminfo* file will grow very large. You may then notice long delays when starting Vim, even for files not related to the large file, because Vim must process the *.viminfo* file each time it starts up.
>
> We recommend specifying some sane but useful limit. We use 50.

/n

> The number of search-pattern history items to be saved. If not specified, Vim uses the value in the `history` option.

:n

> The maximum number of commands from the command-line history to save. If not specified, Vim uses the value in the `history` option.

'n

> The maximum number of files for which Vim maintains information. If you define the `viminfo` option, this parameter is required.

Here is what Vim saves in the `viminfo` file:

- Command-line history
- Search string history
- Input-line history
- Registers
- File marks (e.g., a mark created by `mx` is saved, and you can move the cursor to mark *x* when reediting the file)

- The last search and substitute patterns
- The buffer list
- Global variables

This option is really handy for sustaining continuity across sessions. For example, if you edit a large file in which you are changing a pattern, the search pattern is remembered as well as where the cursor is positioned in the file. To continue searching in a new session, you need only type n to move to the next occurrence of the search pattern.

The mksession Command

Vim saves all information specific to a session with its :mksession command. The sessionoptions option contains a comma-separated string specifying which components of a session to save. This way of saving session information is much more comprehensive but much more specific than viminfo. Saving session information this way is specific to all of the files, buffers, windows, and so on, in the current session, and mksession saves the information so that the entire session can be reconstructed. All of the files being edited and all of the settings for all options, even window sizes, are saved so that reloading the information brings back an exact re-creation of the session. Contrast this with viminfo, which only restores editing information on a per-file basis.

To save a session this way, enter:

    :mksession [*filename*]

where *filename* specifies a file in which to save the session information. Vim creates a script file that, when executed later with the source command, reconstructs the session. The default filename, if none was specified, is *Session.vim*. So if you save a session with the command:

    :**mksession** mysession.**vim**

you could later reestablish the session with the command:

    :source mysession.**vim**

Here is what you can save from a session, and the parameter in the sessionoptions option to save it:

blank
    Empty windows.

buffers
    Hidden and unloaded buffers.

curdir
   The current directory.

folds
   Manually created folds, opened/closed folds, and local fold options.

   It wouldn't make any sense to save anything but manually created folds. Automatically created folds will be automatically re-created!

globals
   Global variables, which start with an uppercase letter and contain at least one lowercase letter.

help
   The help window.

localoptions
   Options defined locally to a window.

options
   Options set by `:set`.

resize
   Size of the Vim window.

sesdir
   The directory in which the session file is located.

slash
   Backslashes in filenames replaced with forward slashes.

tabpages
   All tab pages.

   If you do not specify this in the `sessionoptions` string, only the current tab session is saved as a standalone entity. This gives you the flexibility of defining sessions either at the tab level or globally across all tabs.

unix
   Unix end-of-line format.

`winpos`
> The position of Vim's window on the screen.

`winsize`
> The size of buffer windows on the screen.

So, for example, if you want to save a session to retain all information for all buffers, all folds, global variables, all options, window size, and window position, you would define the `sessionoptions` option with:

```
:set sessionoptions=buffers,folds,globals,options,resize,winpos
```

What's My Line (Size)?

Vim allows lines of virtually unlimited lengths. You can have them either wrap onto multiple screen lines, so that you can see them all without horizontal scrolling, or you can display the beginning of each line on one screen line and scroll to the right to see hidden parts.

If you prefer one line of text per screen line, turn off the `wrap` option:

```
:set nowrap
```

With `nowrap`, Vim displays as many characters as the screen width permits. Think of the screen as a view port or window through which the wide line is viewed. For example, a 100-character line contains 20 characters too many for a screen that is 80 columns wide. Depending on what character is displayed in the screen's first column, Vim determines which characters in the 100-character line are not displayed. For example, if the screen's first column is the line's fifth character, characters 1–4 are to the left of the visible screen and therefore are invisible—that is, they are not displayed. Characters 5–84 are visible in the screen, and the remaining characters, from 85 to 100, are to the right of the screen and are also invisible.

Vim manages how the line is displayed as you move left and right through the long line. Vim shifts the line left and right a minimum of `sidescroll` characters. You can set its value as follows:

```
:set sidescroll=n
```

where n is the number of columns to scroll. We recommend setting `sidescroll` to 1, because modern PCs easily provide the processing power necessary to smoothly shift the screen one column at a time. If your screen slows down and response times lag, you may need to increase the value to something higher to minimize the screen redraws.

The `sidescroll` value defines a *minimum* shift. As you probably expect, Vim shifts far enough to complete any motion commands. For example, typing `w` moves the cursor to the next word in the line. However, Vim's treatment of the movement is a

bit tricky. If the next word is partially visible (on the right), Vim moves to the first character of that word but does not shift the line. The next w command shifts the line to the left far enough to position the cursor over the first character of the next word, but only far enough to expose this first character.

You can control this behavior with the sidescrolloff option. sidescrolloff defines the minimum number of columns to maintain to the right and left of the cursor. So, for example, if you defined sidescrolloff to be 10, Vim maintains at least 10 characters of context as the cursor nears either side of the screen. Now when you move left and right on a line, your cursor will never get closer than 10 columns from either side of the screen, as Vim shifts enough text into view to maintain that context. This is probably a better way to configure Vim in nowrap mode.

Vim provides convenient visual cues with the listchars option. listchars defines how to display characters when Vim's list option is set. Vim also provides two settings in this option that control whether to use characters to indicate if there are more characters to the left or right of the visible screen for long lines. For example:

```
:set listchars=extends:>
:set listchars+=precedes:<
```

tells Vim to display a < in column one if a long line contains more characters to the left of the visible screen, and a > in the last column to indicate there are more characters to the right of the visible screen. Figure 13-8 shows an example.

Figure 13-8. A long line in nowrap mode (WSL Ubuntu Linux, color scheme: morning)

In contrast, if you prefer to see a whole line without scrolling, tell Vim to wrap the lines with the wrap option:

```
:set wrap
```

Now the line appears as in Figure 13-9.

Figure 13-9. A long line in wrap mode (WSL Ubuntu Linux, color scheme: morning)

Very long lines that can't be entirely displayed on the screen are displayed with the single character @ in the first position, until the cursor and file are positioned in such a way that the line can be displayed completely. The line in Figure 13-9 appears as shown in Figure 13-10 when it is near the bottom of the screen.

```
   53 $
   @
   @
   @
```

Figure 13-10. Long line indicator (WSL Ubuntu Linux, color scheme: morning)

Finally, Vim lets you make space characters visible. Sometimes for a quick visual check we define the period as the representation of a space, and we later remove it. To make space characters visible, add the period to `listchars` like so:

```
:set list
:set listchars+=space:.
```

To remove it:

```
:set list
:set listchars-=space:.
```

Abbreviations of Vim Commands and Options

There are so many commands and options in Vim that we recommend learning them by name first. Almost all commands and options (at least any that have more than a few characters) have some associated short form. These can save time, but *be sure* you know what you're abbreviating! We have had some embarrassing and unexpected results using short forms thought to be one thing that turned out to be something quite different.

As you become more experienced and develop your favorite subset of Vim commands and options, using some of the abbreviated forms for commands and options saves time. Vim typically tries for Unix-like abbreviations for options and allows for the shortest unique initial substring for command abbreviations.

Some abbreviations for common commands include:

```
n      next
prev   previous
q      quit
se     set
w      write
```

Some abbreviations for common options include:

```
ai  autoindent
bg  background
ff  fileformat
```

```
ft  filetype

ic  ignorecase

li  list

nu  number

sc  showcmd (not showcase—there is no such option)

sm  showmatch

sw  shiftwidth

wm  wrapmargin
```

Short forms for commands and options save time when you know your commands and options well. But for scripting and setting up sessions with commands in your *.vimrc* or *.gvimrc* files, you're more likely to save time in the long run by sticking with full command and option names. Your configuration file and scripts are easier to read and debug when you use full names.

> Note that this is not the approach taken with the suite of Vim script files (`syntax`, `autoindent`, `colorscheme`, etc.) in the Vim distribution, though we take no issue with their approach. We just recommend, for ease of managing your own scripts, that you stay with full names.

A Few Quickies (Not Necessarily Vim-Specific)

We now offer several techniques—some of which are offered by basic `vi` as well as Vim—that are worth remembering and having handy:

A quick swap

A common typing error is to enter two characters in the wrong order. Position the cursor over the first wayward character and type `xp` (delete character, put character). (This was mentioned earlier, in the section "Transposing two letters" on page 33.)

Another quick swap

Got two lines you'd rather swap? Position the cursor on the top line, and type `ddp` (delete line, put line after current line).

Quick help

Don't forget about Vim's built-in help. A quick tap on the F1 function key splits your screen and displays the introduction to the online help. (This is true of `gvim`. If you're in a terminal emulator, that program may preempt F1 for itself.)

What was that great command I used?

In its simplest form, Vim lets you access recently executed commands by using the arrow keys in the command line. Moving up and down with the arrow keys, Vim displays recent commands, any one of which you may edit. Whether or not you edit a command from Vim's history, you can execute the command by pressing the ENTER key.

You can get even more sophisticated by invoking Vim's built-in command history editing. Do this by entering CTRL-F on the command line. A small "command" window opens up (with the default height of 7) in which you can navigate with normal Vim motion commands. You can search as if in a normal Vim buffer and make changes.

In the command editing window, you can easily find a recent command, modify it if necessary, and execute it by pressing ENTER. You can write the buffer to a filename of your choice to record the command history for future reference.

> For a more detailed exercise using the command-line history window as a tool, see the section "Introducing the history windows" on page 341.

A bit of humor

Try entering the command:

```
:help "the weary"
```

and read Vim's reply.

More Resources

Helpful online resources include HTML renditions of Vim's built-in help for the two most recent major Vim releases, Vim 7 and Vim 8.[3]

Additionally, *https://vimhelp.org/vim_faq.txt.html* is a Vim FAQ list. It doesn't link questions to answers, but it is all on one page. We recommend scrolling down to the section with the answers and scanning from there.

The official Vim page used to host tips on Vim, but because of problems with spammers, the administrators moved the tips to a wiki where spam is more easily managed.

---

3 Thanks and a tip of the hat to Carlo Teubner, who maintains the current Vim HTML documentation.

Some Vim Power Techniques

This chapter demonstrates some of the lessons learned over many (too many?) years learning and using Vim. Tweaking some defaults and remapping default commands make hours-long daily use of Vim much more pleasurable. We hope that these ideas and techniques will nudge you to new ideas and cause you to create your own power techniques.

Several Convenience Maps

Command mode in Vim has enough actions and commands that hardly any keys are available to freely use without changing the default behavior. Fortunately, Vim mostly gets it right, and while you may or may not initially agree with its choices, you almost always quickly develop muscle memory for the commands you like to use.

We have picked some convenient alternatives by replacing mappings that either didn't make sense, were redundant and served by more than one key, or were better served by simply mapping them to something more useful.

Exiting Vim Simplified

Way back in the section "Saving and Exiting Files" on page 74, we introduced the several options for exiting vi and Vim. As was demonstrated in Figure 5-1, not everyone gets it the first time. Indeed, "How to exit the Vim editor?" is one of the most popular questions on Stack Overflow, having been asked by more than one million users!

We can easily reduce the three or four keystrokes needed to exit Vim to just one, using these simple key remappings:

```
:nmap q :q<cr>
:nmap Q :q!<cr>
```

`:nmap` is a variant of the standard ex `:map` command. Vim has multiple such variants. We won't go into detail; instead, see `:help :map-modes`.

These two maps let you exit Vim, either regularly (Q) or forcefully (SHIFT-Q), with just a single keystroke!

Resize Your Window

We like to be able to easily resize windows as needed. GUI Vim implementations make this easy by letting you grab and drag status lines between windows, though as purists, we prefer to avoid switching from the keyboard to the mouse. So we found two adjacent keys, _ (underscore, i.e., SHIFT plus minus sign) and + , that not only are redundant[1] with other more easily accessible keys but, happily, mnemonically match "get smaller" and "get bigger," respectively.[2]

So to the point, we map _ to decrement the focused window size, and we map + as its complement to increment the focused window size. Try this either by entering the following lines (as ex commands) to see the behavior, or by adding them to your *.vimrc* file:

```
map _ :resize -1<CR>
map + :resize +1<CR>
```

Now in any window you can decrease or increase the window with _ or +, respectively. This is useful! And it will come in handy for our discussion later about Vim's command history window.

Double Your Fun

One of us has two more favorite remaps he thinks align better with the general Vim philosophy. That is, when a command character in `vi` command mode is doubled up, it is typically a shortcut to a default or intuitive behavior. For example, `dw` deletes a word, and the doubled d (`dd`) deletes the current line. Similarly, `yy` yanks the current line.

We apply this philosophy to two mappings, making Vim features quickly available with more intuitive keystrokes. These configurations activate Vim's powerful command and search histories, both of which, when activated, appear in a new, horizontally split window.

---

1 The minus-sign key is essentially redundant with k, with the slight difference that it positions the cursor to the first nonblank character, while the + character is completely redundant with ENTER and really can be remapped without losing any functionality.

2 Yes, technically the shifted keys represent *underscore* and *plus*, but the mnemonic is the juxtaposition of the keys, with the one key that *does* have *minus* adjacent to the other key that *does* have *plus*. (Just sayin'.)

Introducing the history windows

Vim has a seemingly little-known feature that we think is one of its most powerful ones: the command-line window. Vim stores histories of your `ex` commands and of your search patterns. These stored commands and patterns are accessed in Vim's command-line window, which consists of a new, short window opened at the bottom of your screen. This window is used for both histories. Commands and search patterns are stored, accessed, and manipulated separately.

You can go to the command-line or search-pattern window in two ways (each). The default Vim behavior uses either CTRL-F or `q:` (a `vi` command mode command) to open the command-line window. For more detailed information, use Vim's help to read about the command-line window:

```
:help c_CTRL-F
```

As with any Vim window, use the regular `:q` command to close it.

We talk shortly about some of the cool things to do in this window. But first, let's make it easier and more intuitive to open it. (CTRL-F and `q:` are easy enough, but really, are they intuitive?)

So let's map our way to the command-line window.

Two colons are better than one

In keeping with "double-*something* does *something*-amplified and hopefully intuitive," we decided that a good fit is to double up the colon, which by itself starts an `ex` command. It makes sense that a double colon, `::`, would be "`ex` amplified" by opening the command-line window.

Remember that we are defining a map and assuming it's invoked in command mode. Further, and importantly, since we are mapping `::` to a sequence with a *q*, let's make it safer by requiring that it not be remapped. This calls for the `:noremap` command.

As with the `:map` command, Vim has multiple variants of the `:noremap` command. Without going into detail here either (see `:help :map-modes`), to map `::` correctly, we want to use the `:nnoremap` command. The command looks like this:

```
:nnoremap :: q:
```

Now let's try it out. Enter this command interactively, or add it to your *.vimrc* file. Then in command mode, quickly type two colons. You should now see the command-line window with the cursor placed on the last line. When visiting from `vi` command mode, this will *always* be a blank line.

Why? There are two different behaviors. If you are in the middle of typing an `ex` command and you enter CTRL-F, Vim opens the command-line history window in

`vi` command mode, with the cursor at the end of the last line, showing the partial command you were entering.

When entering the command-line window from `vi` command mode, there is no `ex` command in progress, so the cursor is on a blank line.

And two slashes are better than one

We find it equally intuitive that a double-slash (`//`) be used to quickly activate the command-line search history window.

The default ways to activate the search history window are `q/` and `q?`. Analogous to the previous section's `::` to activate the command-line window for command history, the same approach makes logical sense for visiting Vim's search-pattern history. Just as `/` initiates a search from command mode, we chose `//` as the intuitive mapping for the search history window. Since `?` activates a backward search, we also provide a mapping for `??`.

As with `::`, which we assumed is invoked in `vi` command mode, we use the `:nnoremap` command. The commands are:

```
:nnoremap // q/
:nnoremap ?? q?
```

How Tall Is Your Command Window?

As an aside, the default height of Vim's command-line window is seven lines. We find ten lines to be more satisfactory and set it in our *.vimrc* file, like so:

```
set cmdwinheight = 10
```

Note that, as discussed earlier, if you've mapped _ and + to expand and shrink your focused window, you can conveniently resize the command-line window.

Now let's try it out. In command mode, quickly type two slashes. You should now see the pattern history window with the cursor placed on the last line, which is the last-used search pattern.

Note that Vim inserts a single character in the leftmost column of the command-line window indicating which mode the window is in, using `:` for command-line history, and `/` or `?` for search-pattern history.

You are in a window of commands (or search patterns). This window is special; there are some `ex` commands that you cannot use. In particular, the commands `:e`, `:grep`, `:help`, and `:sort` are unavailable. Nor can you use the commands that move you to another window, leaving this window open, such as `CTRL-W` `CTRL-W`. While these restrictions do not diminish the power of the command-line window, they do emphasize that the window is a special-purpose one.

Some points to be aware of:

- You can navigate this buffer like any other Vim buffer. Don't be shy! Play around with your favorite Vim commands:

  `:w` *filename*
  > To save the buffer to a file.

  `:r` *filename*
  > To read a file into the command-line buffer.

- You can write/save the contents of the command-line buffer much like any other buffer.
- Going the other way, you can read files into the command-line buffer.

These final two points are important, as they give you the flexibility to save command-line history you may find useful for later. You can then "load up" a file containing commands into a session and selectively find and execute your very best commands.

Moving into the Fast Lane

Now that we've tamed the command-line window, let's do some other interesting things.

Finding a Hard-to-Remember Command

We'll start by discussing different ways to find that *one* command you need to execute.

Directly searching for the command

You know you used a Vim command to save a lot of time, but you don't remember what it was, nor do you even remember if it was very recent. You *do* remember that it was related to converting some *TEST* descriptions to *PROD*. So you type `::`, and

you're ready to find that command. There are various approaches, all leveraging Vim to find Vim commands.

For example, probably the easiest, most direct approach is to simply search backward in the command-line command history buffer. You know you used something with *TEST* and then *PROD* in the same command. Simply search up through the commands with the search:

```
?.*TEST..*PROD
```

and Vim positions the cursor on the first found line matching that regular expression. Now you can simply re-execute that command by hitting ENTER. Vim closes the command-line window and executes the command automatically.

If the first match *isn't* what you're seeking, the `vi` command `n` moves you to the next match. Use `n` as many times as necessary until you find your desired command.

> When searching back through commands in the command-line window, Vim honors your `wrapscan` setting.[3] If you've set `nowrapscan` and there is no occurrence of your pattern between the current line position and the top of buffer (or the bottom, depending on the direction of the search), Vim displays "search hit top…" (or bottom) without finding the pattern.

Filtering the buffer

You may not like searching as described in the previous section for various reasons; there are many similar commands, the commands are scattered, and so on.

You can filter/alter the command-line window buffer the same way you filter text in regular Vim windows. Continuing with the previous example, assume that you're interested in finding versions of commands with *TEST* and then *PROD* somewhere in the commands.

Rather than simply searching up through the buffer, declutter the buffer by inverting a global search (`:vg`) to delete lines *not* matching your target:

```
:vg/.*TEST..*PROD/d
```

Now you have *only* lines matching your target. Choose your match and execute.

Lines deleted in this way stay deleted (inaccessible) until the end of your editing session. But the next time you open the file in Vim, all the saved command lines will be there.

---

3 This also applies to the behavior in the command-line window for search patterns.

Massaging the filter results

Since Vim provides editing in the command-line window itself, it's a natural next step to consider more than simply finding commands and re-executing them. Often the task at hand is similar but not identical to tasks completed earlier whose commands are in the history buffer. However, the task is similar *enough* to use commands from the history buffer with slight modifications.

Continuing again with the previous example, consider that we want to do the same Vim commands but instead of transforming *TEST* to *PROD*, we want to transform *PROD* to *QA*.

As before, start by finding and perhaps filtering candidate commands from the history buffer. Now change *PROD* to *QA*, and change *TEST* to *PROD* (assuming it's that simple a translation). Hit ENTER on the edited history line to execute it, and you're done!

Analyzing a Famous Speech

One of us found the back-and-forth concerning a particular political speech fascinating. There was much said and argued about the speech and what it said and meant. So he had an idea. Why not edit the speech transcript with Vim and filter the transcript by word frequency?

This use case references a very famous and politically charged speech. It is not our intent to make inferences or imply ideology. It is simply an example of how one of us quickly used Vim as a tool to analyze information not normally thought of as the kind of editing one would expect to use Vim for.

To make life easy, the speech is included in the book's GitHub repository (see the section "Accessing the Files" on page 471) in the file *book_examples/famous-speech.txt*.

With the speech in hand, your author invoked `awk` to begin incrementally developing a command, iterating until he achieved the desired end result. Remember that you can replace a range of lines in a buffer with the output of a command performed on that range. In this case, in `vi` command mode he typed:

```
:%!awk 'END { print NR }'
```

knowing that it would simply replace the buffer with the line count of the buffer. This is *not* what he wanted, but it made a good starting point.

Now that the "seed" of a command was in Vim's command history, it was easy to start improving. In the course of less than ten minutes he iterated in Vim's command-line history window as follows (line numbers added for annotation, long lines wrapped to stay on the page):

```
 1  1,$!awk '{ while (i = 1; i <= NF; i++) word[$i]++ } END { print word }'
 2  1,$!/usr/bin/awk '{ while (i = 1; i<= NF; i++) word[$i]++ }
       END { print word }'
 3  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
       END { print word }'
 4  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
                     END { for (words in word) print word[words], words }'
 5  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
                     END { for (words in word) print word[words], words }'
 6  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
       END { for (words in word) print word[words], words }' | sort
 7  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
       END { for (words in word) print word[words], words }' | sort
 8  wq
 9  1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
       END { for (words in word) print word[words], words }' | sort -n
10  g/fight
11  1,$!/usr/bin/awk 'BEGIN { FS= "[,. ]+" } { for (i = 1; i<= NF; i++) word[$i]++ }
       END { for (words in word) print word[words], words }' | sort -n
12  g//
13  g/law
```

You may notice the different invocations of awk in the preceding list. This is an artifact of the author's frequent moves from one computer and OS to another. We have left the variants visible and expect that you would use the one that works in your environment.

It's *very important* to understand that the accumulated incremental commands in the preceding list are the result of iterating on one command. After iterating to completion, the result is these commands in the command-line history window.

For example, after changing line 1 (wrapped):

```
1,$!awk '{ while (i = 1; i <= NF; i++) word[$i]++ }
         END { print word }'
```

to (wrapped):

```
1,$!/usr/bin/awk '{ while (i = 1; i<= NF; i++) word[$i]++ }
                  END { print word }'
```

the next time you visit the command-line command history window, these will be the last two nonblank lines in that window.

Line 2 corrected line 1 to point to the correct location for awk. When executed (since the range was specified as 1,$), Vim replaced the entire buffer with:

```
awk: cmd. line:1: { while (i = 1; i <= NF; i++) word[$i]++ } END { print word }
awk: cmd. line:1:                  ^ syntax error
awk: cmd. line:1: { while (i = 1; i <= NF; i++) word[$i]++ } END { print word }
awk: cmd. line:1:                              ^ syntax error
```

So that was bad. Fortunately, typing u resets the buffer to the original speech transcript.[4]

Remember, type :: to edit your last command to match the next example command above. Line 3 fixed one syntax error. Line 4 yet another. This is a bit embarrassing.

Finally, the next line (line 5, wrapped to fit the page):

```
1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
                  END { for (words in word) print word[words], words }'
```

yields real results! The first few lines in the buffer now look something like:

```
3 weeks.
4 State
1 you've
1 written
25 you're
1 telephone
1 Congress
1 ever,
5 biggest
38 are
```

We now see a line for every word in the speech, with a count preceding that word indicating the number of occurrences. This is cool but a little bit unorganized. So let's sort the results (line 6, wrapped):

```
1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
                  END { for (words in word) print word[words], words }' | sort
```

This is a little better, but a numeric sort is even better…To do that, add the -n option (numeric sort) to sort (we are now at line 9, wrapped):

```
1,$!/usr/bin/awk '{ for (i = 1; i<= NF; i++) word[$i]++ }
                END { for (words in word) print word[words], words }' | sort -n
```

This is a much better result. The last few lines of the buffer now look something like:

```
115 that
125 they
134 you
146 in
167 I
203 a
227 and
265 of
326 to
394 the
```

It's no surprise that the common word *the* is the most frequent.

---

4 Using u makes it easy to iterate on different manipulations in buffers. Once you get the muscle memory for this, it becomes very natural.

Let's do one last iteration. If you'll notice in the next-to-last iteration we just looked at, some of the words still have their attached punctuation. This leads to misleading counts of words like "car", "car,", and "car,", all of which should probably be considered the same word. So for the final change, let's define the field separator to be a regular expression in awk's BEGIN rule (line 11, again wrapped):

```
1,$!/usr/bin/awk 'BEGIN { FS= "[,. ]+" } { for (i = 1; i<= NF; i++) word[$i]++ }
              END { for (words in word) print word[words], words }' | sort -n
```

Note the difference in counts, indicating that we probably have something closer to a true result:

```
144 that
153 in
155 you
168 I
203 a
210
227 and
266 of
328 to
394 the
```


Remember to undo (u) after each iteration in order to pass the original text to the next command.

Our example leaned on awk as the filter *du jour*, but in the spirit of Unix and GNU/ Linux, there are *many* powerful commands that, when applied to a Vim buffer, give equally useful results. Both of us *prefer* awk, but we often use, in no particular order of preference, sed, grep, wc, head, tail, sort, and many others. It's also worth noting that pipes just work and amplify the power of massaging Vim buffers.

While this may seem like a somewhat contrived example, your author did this exercise while discussing said speech, and the results were used to settle "arguments." The time spent iterating from the initial awk command through to the final, refined command was just a few minutes. The results were a touchstone in an energetic discussion about what *was* in the speech. We offer no opinion for that, but emphasize that this *was* a productive and useful exercise in a social setting. Yes, Vim is probably not a common player in social gatherings, but maybe it can be. ☺

Some More Use Cases

As mentioned, the speech example may seem contrived, but it is one of many examples in which we have massaged files to extract useful information without resorting to external tools. Other examples include:

Exported files

One of us tracks his exercise via a few applications. His favorite is Garmin™. Garmin export files are CSV text files (see Figure 14-1 for an example). Consider how you might extract information in a similar fashion to our example.

System log files

We have used the same technique(s) to extract, massage, and format Unix system log files (e.g., */var/log/messages*). While there are many real-time and supplementary tools that *do* monitor and analyze these files (such as Splunk, sometimes it's enough to simply jump in and quickly apply customized commands from Vim's command-line history window.

Vendor log files

Filtering these is similar to filtering system log files.

Figure 14-1. Sample Garmin CSV file

We've mentioned before that it's useful to set the amount of command history Vim saves to a large number, and we thought we'd mention it again here. This is set by the `viminfo` option, which should be defined in your *.vimrc* configuration file:

```
" example
set viminfo='50,:1000
```

Interestingly, when you edit a command in the command-line history buffer, if the editing context could offer a completion, the TAB key pops up a completion menu that, with repeated TABs, cycles through the list. You may also use the arrow keys. The desired match is selected by hitting ENTER. Be careful, as this triggers the execution of the modified command immediately. Figure 14-2 shows completion for partial matches of options. Figure 14-3 presents an example of filename completion. (Command-line completion is discussed in the section "Built-In Help" on page 165.)

Figure 14-2. Example list of completions for a command requiring an argument (in this case, a color scheme)

Figure 14-3. Example list of completions for a command requiring a filename

Also, to ensure that you've saved important commands you've developed and that they don't fall out of the saved commands limit, you may find it useful to curate these commands and store them in text files. These files can then be "loaded" into your command-line history window.

Hitting the Speed Limit

As we've done for command line history, let's consider some approaches to making your search-pattern history more useful. We use the pattern search history window to edit, iterate, and improve searches. And the searches, once tuned, are then easily retrieved.

Recall our key mapping to more intuitively open the pattern search window:

```
:nnoremap // q/
:nnoremap ?? q?
```

Vim's search-pattern window is really the same buffer as the command-history window; you can have only one active any time. The only differences are the content loaded into the window and the actions taken by hitting ENTER on any line in the buffer (execute an ex command versus search for a pattern). Thus, the same features

exist as described earlier. You can search for any pattern that you've searched with before. Yes, we're searching for searches, which is a little meta. If your search-pattern history is large enough, it's likely to have effective searches you've used previously.

Within this window, you use the normal Vim editing features to navigate, modify, and execute searches from past searches.

Just as earlier we incrementally built a useful filter by iterating on a command using the command-line history window, we can use the same technique and hone searches by incrementally building search patterns.

Vim uses regular expressions for searches, which can be quite complex. Consider the example that a file may contain names of production executables, which are named by convention to identify their role. For example, a production executable may be named by convention where period-delimited fields describe attributes of that executable (e.g., `production.accounting.receivables.east.rollup`). The requirement is that the first field be one of *production*, *test*, or *devel*, and that the entire name comprises five fields.

We won't go into the same detail as in the development of the earlier command (in the section "Moving into the Fast Lane" on page 343), but it's simple enough to start by looking for a line containing *production*:

```
/production/
```

That would find all lines containing any instance of *production* anywhere in the line. A quick `//` takes us to the search-pattern history window, and a quick edit to insert a required delimiter "`.`" narrows the search (note that we've removed the slashes since they do *not* appear in the search-pattern history window):

```
production\.
```

Eventually, a viable result could look something like:

```
\(production\|test\|devel\)\(\.[[:alnum:]_]*\)\{3}\.[[:alnum:]_]\{1,}
```

We once used a pattern similar to this and created a `match` command in our *.vimrc* file to automatically highlight executable names while editing files, supplementing the normal syntax highlighting.

> We leave it to you to decode the final regular expression. The point is less about the regular expression and how it works, and more about the means to the end: how to develop a powerful regular expression using Vim's search-pattern history window.

Just as in the command-line history example, you can curate and save favorite regular expressions that can be loaded later into the search-pattern window buffer.

Enhancing the Status Line

Vim, for some reason, provides a status line lacking in much useful information (see Figure 14-4).

Figure 14-4. The default Vim status line

Without going into great detail, and providing only minimal explanation (see `:help statusline` for the full story), the following *.vimrc* line provides enhanced information about the current file:

```
set statusline=%<%t%h%m%r\ \ %a\ %{strftime(\"%c\")}%=0x%B\ \ line:%l,\ \ col:%c%V\ %P\ %v
```

See Figure 14-5 for an example of a status line using these settings.

Figure 14-5. Elbert's status line

Elbert's *.vimrc* file, from which this example comes, is available in the book's GitHub repository; see the section "Accessing the Files" on page 471.

Here's a brief explanation of the built-in flags used in the example. Flags all start with the % character:

`%a`

Argument list status. For example, if Vim is editing the fourth file of eight, the status line would display (4 of 8).

`%B`

Hexadecimal representation of the character under the cursor.

`%c`

The current column number.

`%h`

Buffer "help" flag (not shown in this case, since we're not editing a help file).

`%l`

The current line number.

`%m`

The modified flag ([+] if the buffer has been modified; not present otherwise).

`%P`

> The current location in the buffer as a percentage.

`%r`

> The read-only flag ([`RO`] if the buffer is read only; otherwise not present).

`%{strftime…}`

> The results of running the command within braces (in this case, `strftime`). Passing `%c` asks for the standard date and time.

`%t`

> The current filename (the final component of the filename, equivalent to the output of *basename*(1)).

`%v`

> The type of the file being edited. This is more than just inspection of the filename for an extension. Vim detects the type of a file based on its content. While we cannot prove it, it appears that Vim uses a mechanism similar or identical to that of the `file` command. (See the *file*(1) manual page, if interested.)

`%V`

> The current virtual column number.

`%=`

> Center information around this anchor (everything before is left-justified; everything after is right-justified).

`%<`

> Truncate here if the status line is too long.

Summary

We hope that the material presented in this chapter whets your appetite for further exploration of Vim's capabilities. You'll find that there's always something more to learn about Vim. Doing so will make your work easier and more productive.

Vim in the Larger Milieu

Part III takes a step back to look at the bigger picture, looking at Vim's role in the larger software development and computer usage worlds. It then closes off the book with a short epilogue. This part contains the following chapters:

Vim as IDE: Some Assembly Required

Although `vi` is a general-purpose text editor, from Day One it was also a programmer's text editor. It has multiple features for making programming easier, particularly programming in C. (Consider the `showmatch` option, the automatic indentation features, and in particular the `ctags` facilities, as well as the facilities for maneuvering within `troff` documentation.)

Unsurprisingly, Vim continues in this tradition, but unlike `vi`, it itself is programmable, and in particular it supports *plug-ins*, the ability to load new code and add features directly into the editor.

As with many of the popular scripting languages, this *extensibility* has led to an explosion of new features and facilities for use with Vim—many more than any one person could have ever created working alone.

Also not surprisingly, a large percentage of these plug-ins are aimed at making programming and software development with Vim much easier.

In this chapter we look (briefly!) at plug-in managers and at some of the more interesting and popular plug-ins for use in software development.

Be aware, however, that the universe of Vim plug-ins is very large. Thorough coverage of all the possible plug-ins would require a separate book—one much bigger than this one is! Therefore, our treatment here involves much less hand-holding than do other chapters in the book; please bear this in mind as you read.

Plug-In Managers

Besides plug-ins that actually do something, there are also plug-ins that manage other plug-ins. Their job is to load and initialize plug-ins, and to make it easy for you

install and use plug-ins without having to manually download them or add a lot of plug-in-specific code to your *.vimrc* file.

Vim has its own plug-in manager, accessed with the `:packadd` ("package add") command. You can use this command with the plug-ins that come standard with Vim, or for any other plug-ins that match the criteria given by `:help packadd` (which we won't get into here). We show one of these standard plug-ins later, and we encourage you to check out the others that come with Vim.

One of the most popular plug-in managers is called Vundle (short for "Vim bundle"). The website has "quick start" instructions, which we attempt to summarize here, assuming a GNU/Linux or other POSIX-style system:

1. Make sure you have Git and `curl` installed on your system.

2. Save a copy of your *.vimrc* file and *.vim* directory someplace safe, just in case.

3. Clone Vundle directly into its place:

   ```
   git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
   ```

4. Configure your plug-ins. This is what your *.vimrc* should look like (or you can copy/paste from the Vundle home page); we've omitted some commentary to keep this short:

   ```
   set nocompatible              " be iMproved, required
   filetype off                  " required

   " set the runtime path to include Vundle and initialize
   set rtp+=~/.vim/bundle/Vundle.vim
   call vundle#begin()
   " alternatively, pass a path where Vundle should install plugins
   "call vundle#begin('~/some/path/here')

   " let Vundle manage Vundle, required
   Plugin 'VundleVim/Vundle.vim'

   " The following are examples of different formats supported.
   " Keep Plugin commands between vundle#begin/end.
   " plugin on GitHub repo
   Plugin 'tpope/vim-fugitive'
   ...

   " All of your Plugins must be added before the following line
   call vundle#end()             " required
   filetype plugin indent on     " required
   " To ignore plugin indent changes, instead use:
   "filetype plugin on
   "
   ...
   "
   " see :h vundle for more details or wiki for FAQ
   " Put your non-Plugin stuff after this line
   ```

5. Pick and choose your plug-ins, and place them between the calls to `vundle#begin()` and `vundle#end()`. (This is the hard part. ☺)

6. Install the plug-ins you've listed in your *.vimrc* file. You can do this either with the `:PluginInstall` Vim command or from the command line:

   ```
   vim +PluginInstall +qall
   ```

   This launches Vim, installs the plug-ins, and then quits. Run `:PluginInstall` every time you add new plug-ins to your *.vimrc* file; Vundle then handles the downloading and installation of the plug-in for you.

Finding Just the Right Plug-In

There are thousands of Vim plug-ins. Many (maybe even most) of them are hosted on GitHub, but not all of them are. Locating the right plug-ins for what you need done could become a daunting task. (See Figure 15-1.)

Figure 15-1. GitHub search for Vim plug-ins; 8,852 results and counting

Fortunately, you're not the first one to notice this. The people at Vim Awesome have done amazing work in scavenging the internet for plug-ins and collecting information about them all in one place. See Figure 15-2.

Figure 15-2. Vim Awesome

Not only can you search for plug-in information at Vim Awesome, but you can also set up your own private copy of their database. The source code and instructions for doing so are at *https://github.com/vim-awesome/vim-awesome*.

Why Do We Want an IDE?

An integrated development environment (IDE) is just what it sounds like—a single environment that provides everything you need to do software development. There are many such environments, both commercial and open source. If you're a software developer, it's likely that you are familiar with at least one.

IDEs generally provide at least the following features:

- Text editing (of course).
- Viewing and navigating the software project's file tree.
- Navigation among source objects (e.g., going from a function call to the definition of the function).
- Integration with one or more source code control systems. For our purposes, integration with Git is what we want.
- Text completion during typing. For example, as you type the name of a function, the IDE shows you what parameters are expected.

- Semantic error highlighting. If you have a semantic error in your program (such as an undeclared variable), the IDE highlights the error, often by drawing a colored wavy line underneath it.

- Integrated debugging. The IDE shows the source code as the debugger moves around within the source.

Given that many programmers spend large amounts of time working in Vim, it's natural to want Vim to provide features similar to those of IDEs, since they increase productivity. We will see shortly how Vim plug-ins provide these features, and how you can customize Vim into an IDE that suits your personal needs.

Doing It Yourself

The early Unix systems were notable in that they focused on mechanism, not policy. That is, the system provided the user the capability to do many things, without enforcing a single way of working.

Vim demonstrates its Unix heritage in a similar fashion: you have the facilities to build almost anything you want. Of course, this means that you then have to invest the time and effort into learning how to build what you want! The return on this investment is an environment exactly suited to your needs.

Fortunately, as we saw earlier, it's likely that there already exists a Vim plug-in to do just about anything you would need. You just have to go and find it!

In the following subsections we take a look at a very few of the most popular plug-ins for software development. Later on, we give an overview of several all-in-one solutions that focus on turning Vim into an IDE.

As you review this section, remember that this survey just barely scratches the surface, and that there's much more out there!

EditorConfig: Consistent Text Editing Setup

During our research, we came across the EditorConfig project. The project's goal is to define a specification for how different text editors and IDEs should format different kinds of files. For example, for one kind of file you might want all of your editors to indent by four spaces, and for others you might want them to indent with real TAB characters. A single *.editorconfig* file lets you do that. Your editor, no matter what it is, reads this file and then formats your work appropriately.

A large number of IDEs support *.editorconfig* files out-of-the-box. Vim requires a plug-in, which may be found at *https://github.com/editorconfig/editorconfig-vim*. Installation instructions are included. See also *https://www.vim.org/scripts/script.php? script_id=3934* for more information and description.

NERDTree: File Tree Traversal Within Vim

The NERDTree plug-in is a major step forward in making Vim function like a standard IDE. Once installed, you open and close its window with the `:NERDTreeToggle` command. The documentation suggests mapping this to a keyboard sequence such as CTRL-N, like so:

```
map <C-n> :NERDTreeToggle<CR>
```

This command opens a new window on the left side of the screen, showing a standard file tree. Type ? in the NERDTree window to see a list of commands you can use to expand or contract directories and to open files in the current or new windows. The behavior varies depending on whether the current line in the NERDTree window is a file or a directory. Some of the commands are:

?

Toggle display of NERDTree help.

i

Open a file in a new window with the `:split` command.

o

Expand/contract a directory. For a file, open the file in the previous window.

s

Open a file in a new window with the `:vsplit` command.

t

Open the file or directory in a new tab. Tabs were described in the section "Tabbed Editing" on page 233.

T

Silently open the file or directory in a new tab.

There are quite a number of other capabilities. Comprehensive documentation is provided in the file *doc/NERDTree.txt*.

(What? No figure? Patience. See the next section.)

nerdtree-git-plug-in: NERDTree with Git Status Indicators

NERDTree in and of itself is exceedingly useful. However, these days *everyone* uses a source code control system, typically Git. Many IDEs can show the source code control status of a file in their file explorers (modified, not under source code control, subdirectory contains untracked files, and so on). The `nerdtree-git-plugin` plug-in at *https://github.com/Xuyuanp/nerdtree-git-plugin* enhances NERDTree with this capability.

Figure 15-3 shows two editing windows and the NERDTree window with the `nerdtree-git-plugin` plug-in on the left side. In that window we see that the *atomtable* directory is untracked (not checked into Git), and that the *support* directory contains a modified file, as does the *helpers* directory. Other icons (not shown here) indicate the modified/untracked status of a given file. All of this makes the Git status of your project's files easily discernible.

Figure 15-3. NERDTree with Git indicators

Fugitive: Running Git from Within Vim

Vim users who manage their files with Git end up moving back and forth between their Vim window and their terminal window in order to issue Git commands. The Fugitive plug-in allows you to stay within Vim while working with Git.

The change to make is very simple. Instead of running the `git` command in a terminal emulator, use `:Git` (or even just `:G`) and continue as usual (`:Git add`, `:Git status`, `:Git commit`, etc.). Fugitive places any output from Git into a new temporary buffer, if necessary. When committing a file, you edit the commit message in the current Vim instance.

Quoting from its web page, Fugitive does more than just run the `git` command for you:

- The default behavior is to directly echo the command's output. Quiet commands like `:Git add` avoid the dreaded "Press ENTER or type command to continue" prompt.

- `:Git commit`, `:Git rebase -i`, and other commands that invoke an editor do their editing in the current Vim instance.

- `:Git diff`, `:Git log`, and other verbose, paginated commands have their output loaded into a temporary buffer. Force this behavior for any command with `:Git --paginate` or `:Git -p`.

- `:Git blame` uses a temporary buffer with maps for additional triage. Press enter on a line to view the commit where the line changed, or `g?` to see other available maps. Omit the filename argument and the currently edited file will be blamed in a vertical, scroll-bound split.

- `:Git mergetool` and `:Git difftool` load their changesets into the quickfix list.

- Called with no arguments, `:Git` opens a summary window with dirty files and unpushed and unpulled commits. Press `g?` to bring up a list of maps for numerous operations including diffing, staging, committing, rebasing, and stashing. (This is the successor to the old `:Gstatus`.)

- This command (along with all other commands) always uses the current buffer's repository, so you don't need to worry about the current working directory.

To demonstrate, Figure 15-4 shows the output of `:Git blame` on this chapter.

Figure 15-4. Running `:Git blame` on a window

Moving the cursor to the fourth line (commit ID `afc75e3d`) and hitting ⎡ENTER⎤ brings up the view shown in Figure 15-5.

```
●    afc75e3d1211030286bbd580cfb7465a17f9279e - (fu...me/arnold/work/ora/vi8/bookfiles/.git) - GVIM1    ● ● ✖
File  Edit  Tools  Syntax  Buffers  Window  Help
```

```
▲ tree fda2633bdf5c61fda1dec0ebc1efa6926ca8e9a9
  parent 9ac06c1b356e9c66f52862e1356a9851df5c1a74
  author Arnold D. Robbins <arnold@skeeve.com> Thu Oct 22 20:51:03 2020 +0300
  committer Arnold D. Robbins <arnold@skeeve.com> Thu Oct 22 20:51:03 2020 +0300
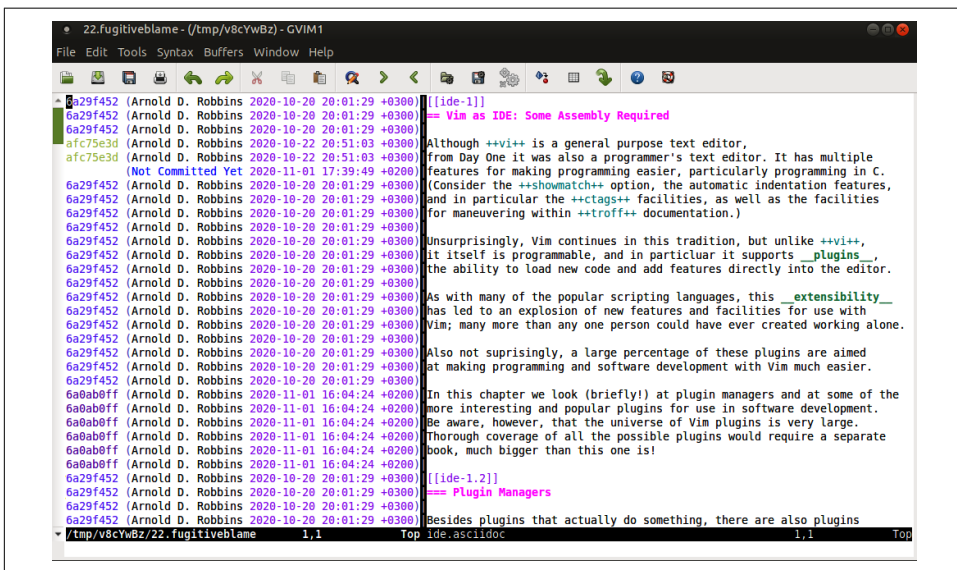
  Continue on ide. Add it to build.


  1+  0- atlas.json-----------------------------------------------------------------
  2+  0- book.asciidoc---------------------------------------------------------------
  diff --git a/ide.asciidoc b/ide.asciidoc
  index 565eaaa..10127d6 100644
  --- a/ide.asciidoc
  +++ b/ide.asciidoc
  @@ -1,8 +1,8 @@
   [[ide-1]]
   == Vim as IDE: Some Assembly Required

  -Although ++vi++ is a general purpose text editor, it was always,
  -from Day One, also a programmer's text editor. It has multiple
  +Although ++vi++ is a general purpose text editor,
  +from Day One it was also a programmer's text editor. It has multiple
   features for making programming easier, particulaly programming in C.
   (Consider the ++showmatch++ option, the automatic indentation features,
   and in particular the ++ctags++ facilities, as well as the facilities
  @@ -19,40 +19,6 @@ Vim; many more than any one person could have ever created working alone.
   Also not suprisingly, a large percentage of these plugins are aimed
   at making programming and software development with Vim much easier.

  -[[ide-1.1]]
  -=== Why Do We Want an IDE?
▼ █                                                          55,1           Top
```

Figure 15-5. Viewing a single commit

There are several "screencasts" demonstrating Fugitive's capabilities. These are worth checking out:

- "A complement to command line git"
- "Working with the git index"
- "Resolving merge conflicts with vimdiff"
- "Browsing the git object database"
- "Exploring the history of a git repository"

We recommend reviewing them and spending some time to come up to speed on this plug-in. After spending just a few minutes with it, we were hooked!

Completion

One of the most powerful features that IDEs provide is *completion*. Depending on the IDE, the programming language you're working in, and possibly various settings, as you type, the IDE offers to help you complete what you're typing. For example, it may fill in the name of a long function for you—or upon your typing the open parenthesis of a function call, it may show the expected types of the parameters, allowing you to

fill in appropriate values. In this section we look in detail at one completion plug-in for Vim and provide pointers to several others.

YouCompleteMe: Dynamic completion and semantic checking

The YouCompleteMe plug-in is very powerful. It provides as-you-type completion and semantic error checking for multiple programming languages. As of this writing, it supports C, C++, C#, Go, JavaScript, Python, Rust, and TypeScript. You may have to install additional software to get things to work for your language.

You can install YouCompleteMe directly from source—instructions are included at the GitHub site. However, you may be able to install it using your system's package manager, and we found this to be the easier way to go.

On one of our Ubuntu GNU/Linux systems, the steps were:

```
sudo apt install vim-addon-manager
sudo apt install vim-youcompleteme
vim-addon-manager install youcompleteme
```

The first command installs `vim-addon-manager`, which is yet another plug-in manager for Vim. There were no conflicts using it with Vundle.

The second command installs YouCompleteMe. The third one installs it into Vim, but only for you as the current user (it's run without `sudo`).

Once installed, Vim begins to offer you completion options in a pop-up window as you type. Press [TAB] to cycle among the options. As you continue to type, YouCompleteMe reduces the number of completion options in the pop-up window. See Figure 15-6.

In and of itself, this is already very cool. But YouCompleteMe goes further by offering semantic analysis of your code. For C and C++ it uses `clangd`, part of the LLVM compiler suite, to continuously recompile your program. It uses different engines for other languages, which you may have to install.

For C and C++, to enable semantic analysis, you have to let YouCompleteMe know how you compile your program. This is done differently depending on how your project is built (Make, CMake, Gradle, etc.).

For Makefile-based projects,[1] this is quite easy. You have to install a simple Python program called `compiledb`, as follows (here too, this is for Ubuntu; other GNU/Linux systems should have an equivalent mechanism):

```
sudo apt install python3-pip
sudo pip3 install compiledb
compiledb make
```

---

1 Real Programmers only use Make.

Figure 15-6. YouCompleteMe's pop-up window

You only need to run `compiledb make` one time (unless you change your compilation options). This creates a *compile_commands.json* file in the top-level directory of your project that YouCompleteMe can use. Once that's done, Vim marks lines with both compilation errors and compilation warnings. See Figure 15-7.

Figure 15-7. YouCompleteMe error indicators

In Figure 15-7, the first problem indicator is highlighted in red, indicating a compilation error. Moving the cursor to that line causes Vim to show the error in the status line. Here, it's an undeclared variable.

The second problem indicator is highlighted in yellow, indicating a warning. Here, the problem is a type mismatch between a `size_t` argument, which is unsigned, and an `int` parameter, which is what `printf()` is expecting.

Note how both problem areas are underlined with a wavy line to show where the errors are (errors in red, warnings in blue). As soon as you fix the errors, the problem indicators go away. This is *extremely* seductive—we wish we'd known about this plug-in years ago!

Configuring YouCompleteMe can be challenging. Putting the following into *~/.ycm_extra_conf.py* may be helpful when working with C instead of C++. Or it may be enough to have `'-std=c99'` in your *compile_commands.json* file. To be honest, we found this part of YouCompleteMe to be frustrating. However, having the semantic warnings is worth it!

```python
import os
import ycm_core

flags = [
  '-fexceptions',
  '-ferror-limit=10000',
  '-DNDEBUG',
  '-std=c99',
  '-xc',
  '-isystem/usr/include/',
  ]

SOURCE_EXTENSIONS = [ '.cpp', '.cxx', '.cc', '.c', ]

def FlagsForFile( filename, **kwargs ):
  return {
  'flags': flags,
  'do_cache': False # True
  }
```

Interestingly enough, YouCompleteMe is not restricted to program source code. It works while editing almost anything, such as a *ChangeLog* file and even the AsciiDoc text of this book!

Other completion and checking engines

There are a number of other completion engines for Vim. Here are some of them:

- The Asynchronous Lint Engine (ALE). This focuses on dynamic linting (semantic checking) of programs, with support for many languages and some support for completion.
- Syntastic. A powerful syntax checking engine with support for many languages, often used with other plug-ins.
- Conquer of Completion. This is a general plug-in with support for many languages and file formats.
- Jedi-vim. This provides Python autocompletion. It's what YouCompleteMe uses under the hood for Python.
- Kite. This plug-in provides AI-based autocompletion for Vim and many other editors and IDEs. It supports Python, C, C++, C#, Go, Java, Bash, and many other languages. It is commercial software, with a free version as well as a for-pay professional version.

Using these engines together with YouCompleteMe doesn't look like it will work. You will want to experiment with them and choose the best setup for you.

Termdebug: Use GDB Directly Within Vim

Beginning with Vim 8.1, it's possible to have a terminal session inside a Vim window. This lets you run programs that interact with a user inside a window. Vim ships with a plug-in named Termdebug that takes advantage of this, letting you run GDB (the GNU debugger) from inside Vim.

To do this, start by editing a file. Then load the Termdebug plug-in with Vim's built-in package manager (`:packadd`) and start it, like so:

```
:packadd termdebug
:Termdebug
```

This splits the screen into three windows. The top window runs GDB. The middle window has the output from the command being debugged, and the bottom window is the source file. You may want to move the source file over to the right, as in Figure 15-8. The section "Moving Windows Around" on page 222 describes how to rearrange the layout of Vim's windows.

In the figure, you see the GDB interactions at top left. On the bottom left is the output from a previous run command where the author mistyped the command (oops!).
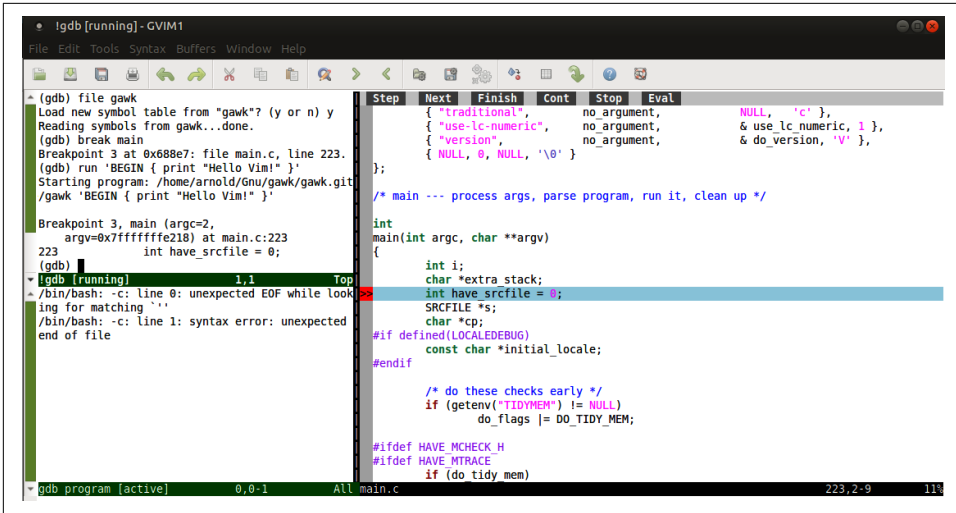
Figure 15-8. Termdebug plug-in running on a file

The righthand window shows the source code, highlighting and marking the line where the breakpoint was hit. Buttons at the top let you continue debugging.

This is a nice integration of GDB with Vim, even better than that provided by Clewn (see the section "The Clewn GDB Driver" on page 384).

If you expect to use GDB frequently, you should put the `:packadd termdebug` command into your *.vimrc* file.

All-in-One IDEs

By now, if you've been adding the plug-ins we've covered, you have Vim providing much of the functionality of an IDE. It's not to be wondered at that many other people have made this journey before us and offered up their own recipes for turning Vim into an IDE.

Here are some of the ones we've discovered, which you may wish to check out. At the very least, they offer you pointers to useful plug-ins that we haven't covered.

Caveat emptor: We have not tried all of these, and of those that we did try, we tried only for a short time. Think of this list as a starting point for your own explorations:

Vim as an IDE
This is something of a tutorial on Vim plug-ins for software development, as opposed to something that is ready to just drop in and go. It's valuable because of the many links to sources for more information. See *https://github.com/jez/vim-as-an-ide*.

vimspector

This is "a multi language graphical debugger for Vim." The focus here is on debugging code, not on being a full-featured IDE. See *https://github.com/pure mourning/vimspector*.

C/C++ IDE

This combination of plug-ins works to provide an IDE for C and C++. Quoting from the web page, you get:

- Automatic download [sic] the latest version of libclang and compile the ycm_core library that YCM needs
- One-step install
- Supported all GNU/Linux
- On-demand loading for faster startup time
- Semantic auto-completion
- Syntax checking
- Syntax highlighting for C++11/14
- Preservation of historical records
- Instantly preview markdown files

The following are for Python:

Python-mode

Quoting from the web page:

The plugin contains all you need to develop python applications in Vim.

- Support Python and 3.6+
- Syntax highlighting
- Virtualenv support
- Run python code (`<leader>r`)
- Add/remove breakpoints (`<leader>b`)
- Improved Python indentation
- Python motions and operators (`]]`, `3[[`, `]]M`, `vaC`, `viM`, `daC`, `ciM`, …)
- Improved Python folding
- Run multiple code checkers simultaneously (`:PymodeLint`)
- Autofix PEP8 errors (`:PymodeLintAuto`)
- Search in python documentation (`<leader>K`)
- Code refactoring
- IntelliSense code-completion
- Go to definition (`<C-c>g`)

Vim and Python—A Match Made in Heaven
> This page from the Real Python folks gives step-by-step instructions for configuring Vim to be a Python IDE.

Vim Upgrade 2017
> This page is similar, providing instructions on configuring Vim and recommending different plugins for day-to-day programming.

Vim as a Python IDE
> Quoting yet again:

>> This project aims to use Vim as a powerful and complete Python IDE. In order to do that, we curated a list of awesome plugins available in the community and provided an automatic installation procedure for this set.

> This project is interesting in that it leaves nothing to chance. It checks out, configures, and builds a specific version of Vim in order to make sure that everything will work. If you want to use Vim for Python development and are OK with the choices it makes for you, this may be the easiest way to go.

chenfjm's VimPlugins
> This is a collection of Vim configuration files that builds an IDE out of 24 different plug-ins. There are terse installation instructions in English, and a tutorial in Chinese. Nonetheless, because it uses so many plug-ins, it's a good source for pointers to additional plug-ins that you may wish to investigate. See *https://github.com/chenfjm/VimPlugins*.

Coding Is Great, but What If I'm a Writer?

There are lots of plug-ins aimed at helping people use Vim for writing, not just software development.

Tomas Fernández presents a nice list in the blog post "Top 10 Vim Plugins for Writers". It's easiest just to quote the article's list of plug-ins with descriptions and links:

vim-pencil
> My favorite writing plugin. Vim-pencil brings a ton of nice things like navigation aids, smarter undo based on punctuation, and proper soft wrapping. (See *https://github.com/reedes/vim-pencil*.)

vim-ditto
> Ditto highlights repeated words in a paragraph, just what I need to avoid repeating words all the time. (See *https://github.com/dbmrq/vim-ditto*.)

vim-goyo
> A Writeroom lookalike for Vim, goyo removes all distracting elements like modeline and line numbers. (See *https://github.com/junegunn/goyo.vim*.)

vim-colors-pencil

An elegant, low contrast colorscheme geared toward writing. (See *https://git hub.com/reedes/vim-colors-pencil*.)

vim-litecorrect

Litecorrect automatically corrects common typing errors like "teh" instead of "the." (See *https://github.com/reedes/vim-litecorrect*.)

vim-lexical

Combined spellchecker and thesaurus. Vim-lexical lets me navigate between spell errors with `]s`, `[s` and quickly find synonyms with `<leader> t`. (See *https://git hub.com/reedes/vim-lexical*.)

vim-textobj-sentence

A plugin for better sentence navigation. I can move between sentences with `(` and `)`, I can cut a sentence with `dis`. Depends on vim-textobj-user. (See *https:// github.com/reedes/vim-textobj-sentence*.)

vim-textobj-quote

This plugin smartly creates "quotes" so I don't have to. (See *https://github.com/ reedes/vim-textobj-quote*.)

ALE

The Asynchronous Lint Engine is a polyglot analysis tool that is not limited to code. It supports a bunch of style checkers like proselint and LanguageTool. (See *https://github.com/dense-analysis/ale*.)

There is more to the article; it's worth reviewing the whole thing.

Conclusion

The Vim Awesome folks got it right: Vim truly is awesome! This chapter has touched only the tip of the iceberg that is the world of Vim plug-ins. We hope that you will set up your own IDE and take full advantage of Vim's powers and abilities, which are far beyond those of mortal text editors.

Just remember to come up for air occasionally as you're exploring all the options available to you. Good luck!

vi Is Everywhere

Introduction

We've described many features that make vi and Vim the powerful editors they are. But vi is more than just an editor. It is a philosophy. It is a way to *think* about words in a different way. It lets us view text as *objects*. These objects, once learned, form an approach to editing far different from "point and click" and "what you see is what you get" (WYSIWYG). Text-as-objects is an interesting abstraction, one so popular that it's rippled into other tools, some of which may surprise you. This chapter introduces some of the common instances of vi-think and some of the less common (but surprisingly useful) instances.

Improving the Command-Line Experience

Just as vi users are power users, their "power" can extend beyond text editing. For years command-line tools (terminal emulators, DOS windows, etc.) provided rudimentary command-line editing and history. More and more, open source contributions have brought dramatic improvements to command-line environments. vi is one of the more popular implementations of command-line history management for many command-line environments.

In Unix the command line is called the *shell*. There are many shells. Some of the most popular are sh (the original Bourne shell), Bash (the GNU Bourne-again shell), csh (the C shell),[1] ksh (the Korn shell), and zsh (Z shell).

---

1 We won't discuss csh, except to mention that the original csh and vi were written by the same person: Bill Joy, when he was a graduate student at the University of California at Berkeley. We also note that almost all shells implement the Bourne shell language, whereas csh is different.

Most but not all modern shells provide `vi`-mode command-line editing, as we are about to see.

Sharing Multiple Shells

Before you test what we're about to present, we *strongly recommend* that you follow the directions we are about to give. We did not do so, and we lost a history file containing almost 8,000 stored history commands!

In the following examples, we describe briefly the option(s) necessary to enable command history editing and then how to navigate your command history with `vi` keystrokes. Since you will necessarily invoke different shells to test the different options, you will create shell instances that each have their own notion of "environment," i.e., variables and behaviors specific to each shell. However, some shells have default values for history files, and when you start or invoke them, they don't bother to override an existing definition of a history file.

For example, if you regularly use `zsh` and invoke a different shell (`ksh`), doing so does not change the value of the history file variable (`HISTFILE`), dutifully recording `ksh` commands in the `zsh` history file. When you exist `ksh`, the existing `zsh` is left dazed and confused, and with a corrupt history file to boot! While this is not the end of the world, if you want the power of history, don't let this happen to you! So here's what you do:

1. In your home directory, create or edit a startup file for each of the shells: `ksh` (*.kshrc*), Bash (*.bashrc*), and `zsh` (*.zshrc*). Make sure you don't overwrite any such file that you already have.

2. In each of these startup files, ensure that you won't lose any valuable history data by adding or verifying that these lines exist:

```
# make BACKSPACE key do what it should do
stty sane

# set command-line editing to vi mode.
set -o vi

# keep history files in a hidden folder please.
myhistorydir=${HOME}/.history
# make the directory, fail silently if it's already there
mkdir -p ${myhistorydir}

# save lots of commands.  computer memory is cheap and reliable.
HISTSIZE=5000
HISTFILESIZE=5000
# save command history in this file.  Note that we incorporate the shell's name
```

```
    # into the file name.  this prevents collisions and corrupt history
    # files inadvertently assigned by different shells (it happens!)
    HISTFILE=${myhistorydir}/.$(basename $0).history
```

The end result of this is that each shell's history is stored in a separate file, based on the name of the shell.

The readline Library

Many GNU and GNU/Linux tools use the `readline` library for interactive input. The `readline` library allows a C (or C++) program to read user input, while providing line editing on the input line.

The Bash Shell

Interactive editing of the shell command line, using either the Emacs or `vi` command set, was first introduced in the Korn shell in the 1980s. The GNU Bourne-again shell (Bash) chose to provide the same features, but built on top of a standalone reusable library called `readline`.

When `readline` is enabled, you get a one-command "window" in your terminal on which you can perform any edits you like using the familiar commands of your favorite text editor, be that either Emacs or `vi`. To enable line editing, you use either `set -o emacs` for Emacs mode or `set -o vi` for `vi` mode. We, of course, prefer the latter. Typically, you would place one or the other of these commands into the *.bashrc* file in your home directory, so that your desired option is always set.

Furthermore, `readline` stores a history of commands that you've executed, so you can move up and down in the list of commands in order to recall and then edit previous commands. For example, k moves up and j moves down in the history list. h and l provide the normal horizontal motions within the current line.

Besides the regular `vi` commands, `readline` provides some additional commands in command mode that perform expansions that are useful on the command line. These are described in Table 16-1.

Table 16-1. Additional vi commands for use in the shell

Command	Action
#	Insert a # at the front of the line, commenting out the line
=	List files with the given prefix
*	Insert the expansion of all files with the given prefix
TAB	Take the preceding prefix, and expand it as far as possible while remaining unique e.g., with a number of `chapterXX` files, and a prefix of ch, TAB would expand ch into `chapter`

Command-line editing in Bash

Taking a real-life example, one of us taught himself Unix by exploring commands in the various Unix command directories (*/bin*, */usr/bin*, */usr/local/bin*, etc.). Leveraging the ability to write complicated commands at the shell prompt, he wrote an on-the-fly script to easily examine various commands with the `man` command, as shown here. The `$` is the main or *primary* prompt; the `>` is the *secondary* prompt issued when Bash knows that the command is not yet complete:

```
$ cd /usr/bin
$ for man in a*
> do
>   printf "\n\n\n$man, look at man page? "
>   read yesno
>   if [ ${yesno:-yes} = "yes" ]
>   then
>     man $man
>   fi
>   printf "\n\nhit enter to continue "
>   read dummy
> done
```

The shell runs a question/answer loop for each file in */usr/bin* starting with the letter *a*. To look at commands starting with other letters, he used Bash's `vi` editing mode by typing ESC K to edit the just-run command line for execution with a new letter. Editing single-line commands in Bash is straightforward. However, multiline commands are a bit messier. The preceding command, when recalled, is presented on a single, very long line, which wraps around on the screen:

```
$ for man in b*; do        printf "n\n\n$man, look at man page? ";
read yesno;        if [ ${yesno:-yes} = "yes" ];        then
man $man;        fi;        printf "\n\nhit enter to continue ";
read dummy; done
```

Note that the shell separator `;` separates all the lines, and note that Bash preserved all of the spacing. To move to each new line, start with `f;` to move to the first semicolon. Using `;` to move to the next occurrence and `,` to move to the previous occurrence, each line is easily found for editing, albeit in a slightly clunky way.

In our example, your author wishes to change the *a* to *b* or to some other character. To do so, he uses his favorite `vi` commands to move to *a** and make the change. Command-line editing, combined with a large stored history, enables him to return to this "script" at any time by locating it in the Bash history and editing it again.

Multiline commands in Bash

Bash does have an option that makes editing multiline commands more pleasant: `shopt -s lithist`. This causes Bash to store multiline commands in the history file as multiple lines, instead of squashed together onto one line with semicolon separators. When enabled, the recalled command looks like this:

```
$ for man in a*
do
  printf "\n\n\n$man, look at man page? "
  read yesno
  if [ ${yesno:-yes} = "yes" ]
  then
    man $man
  fi
  printf "\n\nhit enter to continue "
  read dummy
done
```

You still have to use horizontal motion commands to move around within the recalled text; j and k move up and down in the history list and not within the recalled multiline command. However, you can add additional key bindings to move between physical screen lines. The `readline` commands to do that are `next-screen-line` and `previous-screen-line`.[2] You do that in your *.inputrc* file; see the section "The .inputrc File" on page 380, and the *readline*(3) manual page.

Using Vim to edit Bash commands

If the built-in editor just doesn't do it for you, you can invoke your favorite editor on the command you want to change just by typing v. This puts the contents of your command line into whatever editor is defined by your `EDITOR` environment variable. We expect that this will be `vi` or Vim, of course; however, you can choose any editor you like.

Here's the catch. Bash immediately executes whatever is in the editor buffer when the editor exits. Suppose you enter something like:

```
$ rm -fr /
```

and then type [ESC], and v to enter your editor. If you then decide you don't want to do anything and quit the editor (`:q` or `:q!`), that original text executes, and kaboom!

A safe way to exit Vim and avoid this side effect is to exit with the `:cq` command. This tells Vim to exit with a nonzero return code. That, in turn, tells Bash that an error occurred and that no command should be executed.

Having been burned by this feature, Elbert feels that this is enough justification to consider the Z shell (see further on).

---

2 Thanks to Chet Ramey, Bash's maintainer, for this tip.

Other Programs

Bash isn't the only program that uses `readline`. If it's available when the program is built, GDB (the GNU debugger) uses it, as does GNU Awk (`gawk`) for its built-in AWK debugger. On most GNU/Linux systems, the `ftp` program for interactive internet file transfers also uses it.

Having `readline` integrated with GDB is particularly helpful, as debugging often involves entering repetitive commands, and being able to easily search for and edit previous commands makes debugging much less of a chore. Consider following a chain of "next" pointers in a linked list, for example.

The .inputrc File

> But wait! There's more!
>
> > —Just about every late-night TV commercial ever made

You can customize `readline`'s behavior by putting commands into its initialization file. The `INPUTRC` environment variable points to this file. If `INPUTRC` isn't set, then `readline` looks for a file named *.inputrc* in your home directory. If that file isn't available, `readline` falls back to */etc/inputrc*.

The *readline*(3) man page describes the format and possible contents of this file. The library grows and develops over time, so we won't provide a description of everything here. Instead we present one author's personal *.inputrc* file:

```
set editing-mode vi
set horizontal-scroll-mode On
control-h: backward-delete-char
set comment-begin #
set expand-tilde On
"\C-r": redraw-current-line
```

Here is a brief explanation of what each of these lines does:

`set editing-mode vi`
This turns on `vi` editing mode. The default is Emacs mode. Thus, even in GDB, `ftp`, or any other program using `readline`, the `vi` command set is used.

`set horizontal-scroll-mode On`
This causes `readline` to display only a single line on the screen. Instead of wrapping long lines, the right margin is marked with a `>` character. Moving to the right past the `>` scrolls the line. When the left side of the line goes off the screen, the left side is marked with a `<` character.

`control-h: backward-delete-char`

This causes `^H` (which is usually sent by the BACKSPACE key) to delete characters.

`set comment-begin #`

This causes `readline` to insert a `#` character when the `#` command ("insert comment") is issued. For the shell, this comments out the current line but inserts it into the history for later recall and editing. This is the default for Bash, anyway, but our author's file hasn't changed since 2002!

`set expand-tilde On`

This causes `readline` to do tilde expansion when doing word expansion. This is useful if `readline` is being used with a shell that doesn't do tilde expansion, such as the `rc` shell.

`"\C-r": redraw-current-line`

This causes CTRL-R to redraw the current line. This is useful if output from the system gets intermixed with your input.[3]

Check out the `readline` man page; there are many more options, including some that cause the output to be colored on terminal emulators that support color.

Other Unix Shells

The primeval shell with command-line editing is the Korn shell (`ksh`), originally developed by David Korn while at Bell Laboratories. To enable `vi` mode, use `set -o vi` (this is where Bash got it from). `ksh`'s editor is not based on the `readline` library.[4]

Similarly, the Z shell (`zsh`) has its own `vi` command-line mode; it is somewhat different from that of `ksh` and Bash, and you may have to add additional key bindings if you are used to one of those shells. We discuss the Z shell specifics in more detail in the following section. On the plus side, `zsh` lets you edit multiline commands with ease.

Finally, `tcsh` (the "Tenex `csh`") also provides a `vi` mode, which is enabled with `bindkey -v`. As neither of us are `tcsh` users, we don't have much else to say about it.

---

3 For Emacs users, this is usually your reverse history search key combination. `vi`'s reverse search is simply / (after initiating history search mode by hitting ESC).

4 The Korn shell is still available and being updated.

The Z Shell (zsh)

As mentioned previously, the Z shell has its own command-line mode with a powerful multiline history editor. Consider the example from earlier. The following illustrates the differences in zsh, differences that provide more visual contextual clarity by way of specialized prompts:

```
{elhannah,/usr/bin} for man in a*
for> do
for>    printf "\n\n\n$man, look at man page? "
for>    if [ ${yesno:-yes} = "yes" ]
for if> then
for then>        man $man
for then> fi
for>    printf "\n\nhit enter to continue "
for>    read dummy
for> done
```

And now we get to the real difference in command-line editing! Here we show how zsh presents a multiline command in edit mode after entering ESC K:

```
{elhannah,/usr/bin} for man in a*
do
        printf "\n\n\n$man, look at man page? "
        if [ ${yesno:-yes} = "yes" ]
then
        man $man
fi
        printf "\n\nhit enter to continue "
        read dummy
done
```

You now have a miniature vi session for much more accurate editing.

> While this miniature session lets you insert ("open") new lines of code with o and O, be aware that you must finish the inserted line with ESC. If you hit ENTER, zsh executes the entire block of text immediately.

Keep As Much History As You Can

By now you should have an appreciation for the value of command-line history and editing. In the section "Sharing Multiple Shells" on page 376, where we showed you how to "save your work," we defined *history* variables (related to how many commands to save) thusly:

```
HISTSIZE=5000
HISTFILESIZE=5000
HISTFILE=${myhistorydir}/.$(basename $0).history
```

Think of your dialog, your history of commands issued in your favorite shell, as a large file. Now your command-line history is a living document, with the added benefit of command-line editing for quick and powerful retrieval of long-since-issued commands. The larger the value of HISTSIZE, the further back your shell's memory extends.

Modern computers have an abundance of memory and disk storage. This obviates the old-school need to carefully control your history size. We chose 5,000 as a happy medium. We find this number provides searchable command history measured in years.

The following use case illustrates how powerful command-line history and editing leverage "things we've done before." One of us occasionally works on graphics and videos. However, he often works on other projects and goes many days, weeks, and even months without using any video/graphics applications or commands. But knowing even fragments of commands, or a core application or utility name, he easily finds old examples to refresh his memory on usage *and* is presented with editable commands for immediate productivity. For example, he makes heavy use of the ffmpeg command, which has *many* options and combinations of parameters. Simply by searching ($\boxed{\text{ESC}}$ /ffmpeg $\boxed{\text{ENTER}}$) and iterating with n or N, all saved ffmpeg commands are easily retrieved and available for editing.

Command-Line Editing: Some Closing Thoughts

As you start to use vi-mode command-line editing, keep in mind the notion that your command history is like an editable file. Previous commands are right there at your fingertips (on the HOME keys, no less!). Take advantage of this to improve your mastery of applications by retrieving previous commands ("What was the syntax for that command again?"). Set your saved commands settings *big*! Let your machine do the work. We've given you the starting set of *how-tos*. Now you should explore the man pages and look for the history settings (there are many).

Remember that everything typed in a shell is interpreted as if you are entering a script. Knowing this, leverage some of the shell's behaviors. For example, take advantage of the fact that anything beyond (and including) the # symbol is a comment and is not executed. Adding strategic comments to commands you might want to easily remember gives you one more way of finding old commands. While it was poor security practice, one of us often used # system name passwd appended to an echo command whenever he changed a password for a computer. Then he could easily search his history for the most recent password.

Windows PowerShell

PowerShell is Microsoft's object-oriented command-line environment. It is Microsoft's way to provide quick and powerful automation, forgoing often tedious GUI navigation. The look and feel is reminiscent of Unix shells, and its object-oriented nature extends Unix's "everything is text" philosophy to "everything is an object." The command-line parsing is familiar to any user of the MS-DOS `command.com/cmd.exe` consoles, with some IntelliSense niceties built in. For us, though, it's not enough! Fortunately, you can navigate the PowerShell console with `vi` commands. At the PowerShell prompt, simply enter the command:

```
Set-PSReadlineOption -EditMode vi
```

To make this setting permanent you must add this command to PowerShell's version of *.profile*. As PowerShell has at least six different profile files, we leave the choice of file to you as one of those proverbial exercises for the reader.

Developer Tools

Developers use *many* development tools and have to learn new tools often as they stay current in technology. Having `vi` functionality in development tools makes them more immediately usable and comfortable for developers familiar with `vi` and Vim.

In this section we look at debugging tools with `vi` functionality and at Vim plug-ins for two versions of Microsoft's Visual Studio© IDE.

The Clewn GDB Driver

> Clewn implements full gdb support in the vim editor: breakpoints, watch variables, gdb command completion, assembly windows, etc.
>
> [...]
>
> Clewn is a program controlling vim through the netBeans socket interface, it runs concurrently with vim and talks to vim. Clewn can only be used with gvim, the graphical implementation of vim, as vim on a terminal does not support netBeans.
>
> —The Clewn Project home page

Clewn is an interesting program. It allows you to use Vim to see your source code as you debug in GDB. Clewn controls Vim via the NetBeans interface.[5]

---

5 NetBeans is an open source IDE for Java, JavaScript, HTML5, PHP, C/C++, and other languages. Vim's NetBeans interface allows it to be used as the editor within NetBeans.

To use Clewn, start it in a terminal window. Clewn prompts you with the (gdb) prompt and causes gvim to open another window for displaying your source. This is illustrated in Figure 16-1.
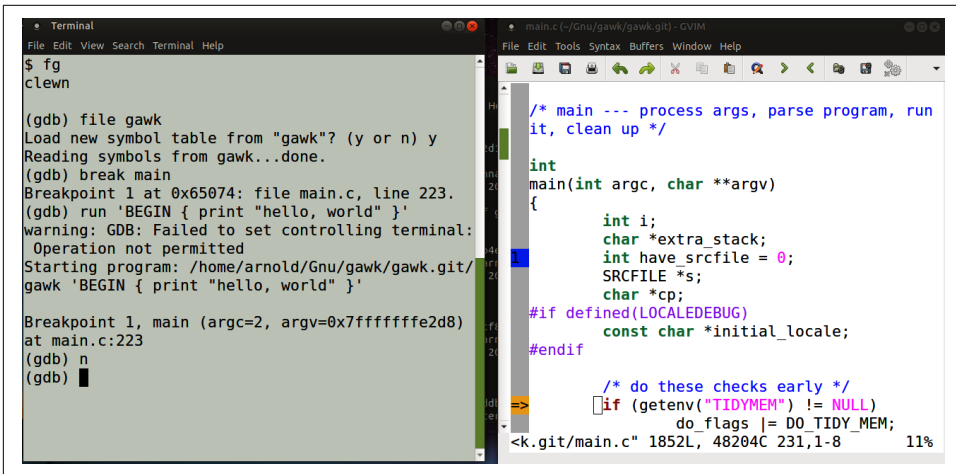
Figure 16-1. Clewn in action

Sadly, Clewn is unmaintained. However, one of us uses it regularly, and it continues to compile "out of the box" and work just fine on current GNU/Linux systems.

The Clewn project home page is at *http://clewn.sourceforge.net*. Source code for Clewn is included in this book's GitHub repository. See the section "Accessing the Files" on page 471 for more information.
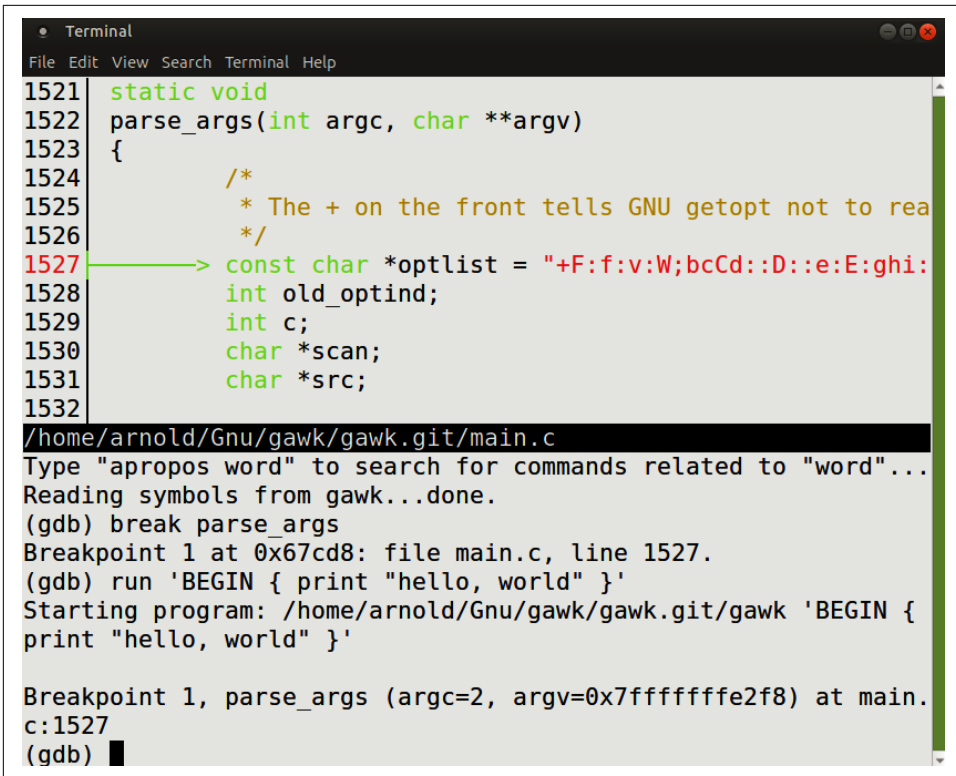
CGDB: Curses GDB

> CGDB is a very lightweight console frontend to the GNU debugger. It provides a split screen interface showing the GDB session below and the program's source code above. The interface is modeled after vim's, so vim users should feel right at home using it.
>
> —The CGDB GitHub page

Debugging seems to be a theme here. This is perhaps not surprising, as vi and Vim are first and foremost programmers' editors, and thus it makes a software development tool more acceptable when it too provides an interface similar to Vim's.

You use CGDB in a terminal emulator window. CGDB splits the screen, showing the (gdb) command prompt in the lower window and your source code in the upper window. What's nice is that the source code is syntax highlighted with different colors. See Figure 16-2.

Figure 16-2. CGDB in action

You use i to move from the command window to the source code window, and ⎡ESC⎦ to move back to the command window. Once in the source code window, you can use Vim searching commands to move around, including the use of regular expressions.

CGDB comes with a full Texinfo manual that explains its use. It provides a nice alternative to Clewn and is considerably nicer than GDB's own built-in text user interface (gdb -tui).

The CGDB project home page is at *https://github.com/cgdb/cgdb*. Check it out!

Vim Inside Visual Studio

Microsoft's Visual Studio is perhaps the world's most widely used IDE. It can be extended with plug-ins, also called *extensions*.

VsVim is an open source extension that provides a Vim emulator for use within Visual Studio. As of this writing, it supports Visual Studio 2017 and 2019, and the project is actively developed and maintained.

If you have to use Visual Studio but would prefer to have Vim-style editing, you should check this one out.

Vim for Visual Studio Code

We discuss Vim plug-ins for Visual Studio *and* Microsoft Visual Studio Code. That is because they are different. Although they are often conflated to be the same application, they are not, hence, different plug-ins.

Visual Studio Code: A quick introduction

Visual Studio Code is Microsoft's lightweight version of their flagship IDE, Visual Studio. It is commonly referred to as *VS Code*. VS Code, like its nonfree commercial big brother, is a powerful development and project-management integration ecosystem. While it offers much of the same functionality, it is different enough that the Vim plug-in options are also different. This is mostly a nit since it is quite easy in either application to add Vim plug-ins.

Obviously, the first thing to do is to download and install VS Code. The installation is very easy and straightforward.

VS Code extensions

Add-ons to VS Code are interchangeably referred to as *extensions* or *plug-ins*. The quickest way to get to VS Code features is to know and use the universal "do this" command. Simply type the command CTRL SHIFT-P for Microsoft Windows and GNU/Linux, or COMMAND SHIFT-P for MacOS. (Interestingly, F1 also does this on all three operating systems.) Begin typing "install extensions" and VS Code will display a drop-down selection list. See Figure 16-3.


```
>install ext

Developer: Reinstall Extension...
Extensions: Disable All Installed Extensions
Extensions: Disable All Installed Extensions for this Workspace
Extensions: Install Extensions
Extensions: Install Specific Version of Extension...
Extensions: Show Installed Extensions
Remote: Install Local Extensions in 'Remote'...
Remote: Install Remote Extensions Locally...
```

Figure 16-3. The VS Code "do this" window

VS Code displays a vertical lefthand-side window summarizing installed extensions and providing search for any installed extension as for well as extensions available in the (vast) plug-in ecosystem. Enter "vim" in the search box and you should see something like Figure 16-4.
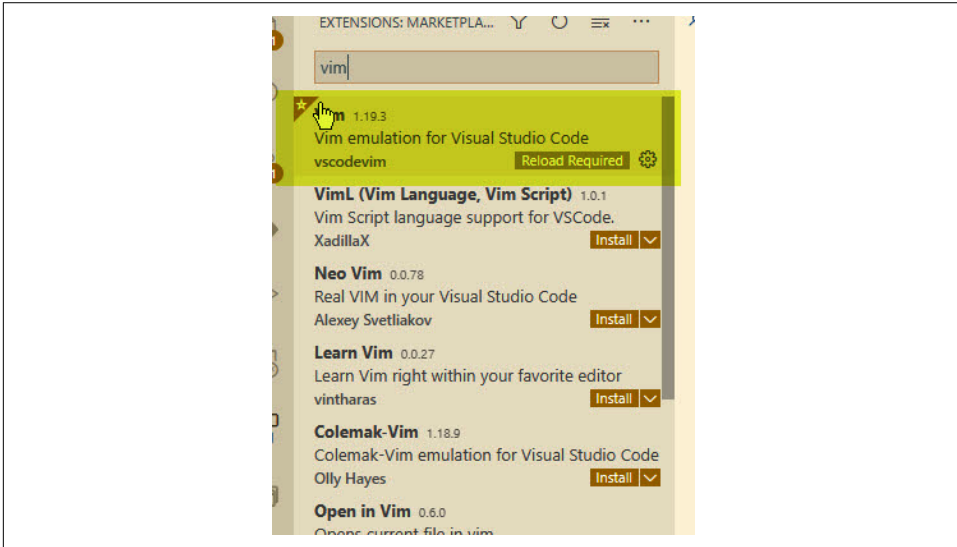
Figure 16-4. Searching for an extension in VS Code

The highlighted item, `vscodevim`, is the plug-in choice. Click the "Install" button. This brings up the dialog box shown in Figure 16-5.
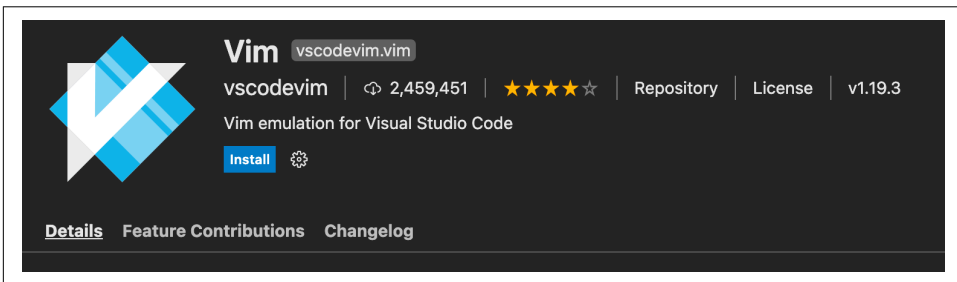
Figure 16-5. Install `vscodevim` dialog

Any time you want to disable or uninstall an extension, find the extension the same way as you did earlier. This brings up the dialog in Figure 16-6. Now click "Disable" or "Uninstall."
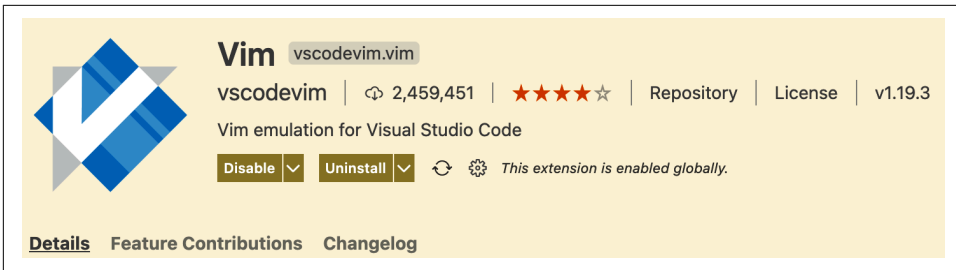
Figure 16-6. Disable or uninstall `vscodevim` dialog

vscodevim settings

To see the available settings for the `vscodevim` plug-in, use the VS Code universal command, ⌐CTRL⌐ ⌐SHIFT-P⌐, and search for *settings*. There will probably be many, but choose "Preferences: Open **Settings** (UI)." See Figure 16-7.

Figure 16-7. Searching for settings in VS Code

Now search in the *settings* dialog, and you'll see that there are almost 100 settings related to `vim`, as shown in Figure 16-8.
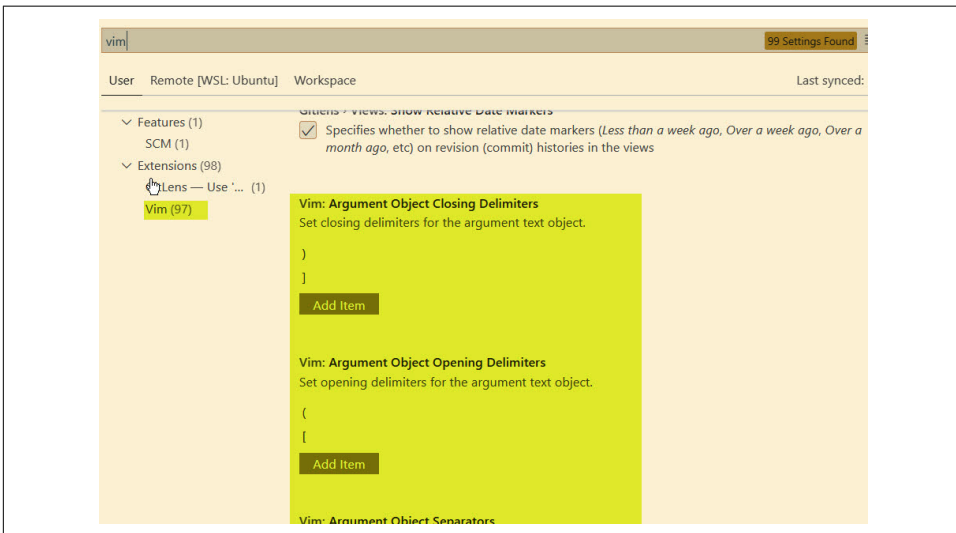
Figure 16-8. VS Code settings for Vim

We leave the preferences to you to curate. Notice that many of the preferences simply provide a link to open VS Code's JSON settings file. Unfortunately, some common Vim customizations require you to get comfortable with editing this file.

For example, as described in the section "Exiting Vim Simplified" on page 339, we like to simplify quitting from Vim by mapping q and Q in vi command mode to :q^M and :q!^M, respectively (quit Vim and *really* quit Vim!).[6] You can't do this in the VS Code extensions preferences. Here is a block of JSON code to create the mapping:

```
"vim.normalModeKeyBindingsNonRecursive": [
        {
                "before": ["q"],
                "commands": [":q"]
        },
        {
                "before": ["Q"],
                "commands": [":q!"]
        },
        {
                "before": ["ctrl+v"],
                "commands": ["Ctrl+Shift+G"]
        }
    ],
```

> Since this activity occurs within VS Code, IntelliSense is quite useful for completing *vim.XX*, where *XX* is the vscodevim setting in play. Note that in the preceding example we use the version "NonRecursive" for normalModeKeyBindings to stop q from being reinterpreted as needing mapping again. This is a common and well-known technique in Vim mapping, defined by prefixing map commands contextually with noremap. (See "Using the map Command" on page 124 for a more complete discussion on mapping vi commands.)

Vim is not just for VS Code

We chose VS Code to discuss Vim plug-ins for IDEs. VS Code has become extremely popular as Microsoft aggressively attends to its development and evolution.

However, VS Code is only one of many IDEs, virtually all of which either offer their own Vim emulation for editing or have readily available Vim-like plug-ins. We have used and verified Vim plug-ins or emulation in NetBeans, Eclipse, PyCharm, and JetBrains. There are many others. Chances are that there is a Vim plug-in for your IDE.

---

6 Remember that ^M is how Vim shows a carriage return, which you enter by typing CTRL-V CTRL-M. See the section "Using the map Command" on page 124.

Unix Utilities

The `vi` abstractions hide in many Unix/Linux utilities that you may be unaware of. Here are some examples of utilities and how `vi` commands improve your efficiency when not editing.

More or Less?

`more` is the original screen-based *pager* program: a program designed to present data one screen at a time. It was developed as part of BSD Unix, in the same timeframe as the original `vi`. Besides presenting file contents, `more` reads standard input if not given any files, making it easy to use at the end of a pipeline.

Some time after `more` became standard, `less`, its name a pun on `more`, was written as an enhanced pager. Today, both are generally available on modern systems.

We think (ironically) that `more` is actually less, and `less` is more. `more` is a legacy Unix tool with basic pagination and interaction, while `less` is almost a streaming editor. `less` lacks real editing features but provides robust navigation similar to Vim.

One of us, on his personal computers (GNU/Linux), always renames `more` to `more_or_less`, and links */usr/bin/less* to */usr/bin/more* (thus making all things `more`, `less`).

If you lack administrative permission or share a system where others may object, you can set up the same thing with an alias that you would add to your personal *.bashrc* file:

```
alias 'more=less'
```

One of our technical reviewers warned that renaming `more` and linking `less` to `more` could cause unexpected behaviors and confusion, possibly affecting the execution of scripts and installers that expect `more`'s original behavior. We agree with the warning, and the advice is worth noting. You should make these modifications with discretion.

For the record, we have made the modifications just described on every GNU/Linux installation since the advent of the `less` command and have never noticed any adverse effects in the system. Nonetheless, your mileage may vary.

Another option to cause `less` to be used instead of `more` (at least most of the time) is to set the PAGER environment variable in your *.bashrc* file:

```
export PAGER=less
```

With that, "the less said, the better." Here are `less`'s Vim-like features:

`b`, `d`

> Go up or down one page, respectively (not available in `more`).

`gg`, `G`

> Go to the beginning or bottom of the input file or stream respectively (not available in `more`).

`/pattern`, `?pattern`, `n`, `N`

> Search forward or backward for *pattern*, and find the next match in the same direction, or in the opposite direction.

`v`

> Open the current file in the editor named by the `EDITOR` environment variable. This works only for a file, not for standard input.

Configuring less's display

A somewhat obscure feature of `less` is the ability to configure how it displays text. For most uses this isn't interesting, but now that we've exchanged `less` for `more`, a real difference and a nice enhancement changes how you'll see manual pages by issuing the following commands. Try viewing a few manual pages before and after to appreciate the visual effect (e.g., `man man`):

```
# variables and dynamic settings to improve less
export LESS="-ces -r -i -a -PM"
# Green:
export LESS_TERMCAP_mb=$(tput bold; tput setaf 2)
# Cyan:
export LESS_TERMCAP_md=$(tput bold; tput setaf 6)
export LESS_TERMCAP_me=$(tput sgr0)
# Yellow on blue:
export LESS_TERMCAP_so=$(tput bold; tput setaf 3; tput setab 4)
export LESS_TERMCAP_se=$(tput rmso; tput sgr0)
# White:
export LESS_TERMCAP_us=$(tput smul; tput bold; tput setaf 7)
export LESS_TERMCAP_ue=$(tput rmul; tput sgr0)
export LESS_TERMCAP_mr=$(tput rev)
export LESS_TERMCAP_mh=$(tput dim)
export LESS_TERMCAP_ZN=$(tput ssubm)
export LESS_TERMCAP_ZV=$(tput rsubm)
export LESS_TERMCAP_ZO=$(tput ssupm)
export LESS_TERMCAP_ZW=$(tput rsupm)
```

This file is available in the book's GitHub repository. See the section "Accessing the Files" on page 471.

There are many more features. We leave the rest for you to explore. Hint: At the `less` prompt, type `h` to get help.

screen

`screen` is a terminal session multiplexer providing multiple concurrent working sessions inside one terminal window. With terminal emulators available today with multiple sessions, usually implemented across tabs, what makes `screen` worthwhile? We will answer that question after presenting how to use `screen`, and in particular after showing a useful sample configuration file. So patience, please.

`screen` provides faithful terminal behavior for its sessions and thus requires you to use a special prefix character to activate its commands and functions. The default is CTRL-A. For example, to display a summary of most available `screen` commands, the `screen` command is ?. So you must type CTRL-A ? to display the summary. Keep this in mind when experimenting with `screen`.

> The following presentation assumes the use of a GNU/Linux or Unix system and that the `screen` application is available. Verify your system has `screen` with the `type` command from the shell prompt:
>
> ```
> $ type screen
> ```
>
> A response like `screen is /usr/bin/screen` is good. If you don't get such a response, install `screen` with your package manager. Start at *https://www.gnu.org/software/screen* if you want (or need) to build `screen` from source.

Getting started with screen

`screen`, like many Unix applications, reads an optional user configuration file to set options and define sessions. The `screen` configuration file, *.screenrc*, lives in your home directory. To help follow the discussion, you can either copy the *.screenrc* file from the book's GitHub repository or create the file yourself and put the following lines in it:

```
startup_message off
defscrollback 20000

# Help screen, key bindings:
bindkey -k k9 exec sed -n '/^# Help/s/^/^M/p;/^# F[1-9]/p' $HOME/.screenrc
# F2 : list windows:
bindkey -k k2 windowlist
# F3 : detach screen (retains active sessions -- can reconnect later):
bindkey -k k3 detach
# F10: previous window (e.g., window 4 -> window 3):
bindkey -k k; prev
# F11: next window (e.g., window 2 -> window 3):
bindkey -k F1 next
# F12: kill all windows and quit screen (you will be prompted):
bindkey -k F2 quit
```

```
screen -t "edits for chapter 1"
screen -t "manage screen captures"
screen -t "manage todos"
screen -t "email and messages"
screen -t "git status and commits"
screen -t "system status"
screen -t "remote login to NAS" ssh 10.0.0.999
screen -t "solitaire"

select 1
```

Now start `screen`:

```
$ screen
```

You should see a normal-looking session, i.e., a command prompt in a window. The last line in the preceding code block, `select 1`, tells `screen` to start your interaction in the first window or session (defined as "edits for chapter 1"). The beauty of `screen` is that you are in a session, one of eight defined in the configuration file, each completely independent from the others.

`screen` has many ways to select sessions and navigate among them. We want to illustrate the `vi`-like navigation.

> `screen` uses single-character commands for most actions. Each of these single-character actions must be preceded by `screen`'s leader command-character, which by default is $\boxed{\text{CTRL-A}}$.

The screen menu

`screen`'s command to display a list of available sessions is the double-quote ("). So, keeping the preceding tip in mind, you display the session menu with the two keystrokes, $\boxed{\text{CTRL-A}}$ $\boxed{"}$. When you do, you should see something like Figure 16-9.


```
         ~/.git/learning-the-vi-and-vim-editors-8e — screen ▸ zsh — ttys001
Num Name

 0 edits for chapter 1
 1 manage screen captures
 2 manage todos
 3 email and messages
 4 git status and commits
 5 system status
 7 solitaire
```

Figure 16-9. Available sessions in screen

While it's only very basic, you can select a session line with the standard line-up and line-down `vi` commands, `k`, and `j`. That's not a lot of `vi`, but obviously in this

context anything else is not really necessary. The point is that this is a philosophy of movement, and the `screen` developers chose `vi`.

Navigating your session's output

There is much more `vi` interaction in the `screen` sessions themselves. `screen` buffers text during your session(s), and you can navigate through this saved text using `vi` commands by typing CTRL-A ESC. By default, `screen` stores only about 100 lines of text. As you can see in our configuration above, we changed that default to 20,000. This is much more reasonable on modern systems. Really, make sure you set this. One hundred cached lines of text isn't much at all.

Each buffer is maintained separately per session. Thus, in addition to command-line history that you can search, edit, and re-execute, you have `vi`-like access to both your commands *and* your commands' outputs!

The basics for searching a `screen` buffer start with `vi`-basics. After typing CTRL-A ESC. you can search the buffer with motion commands (line up is k, line down is j, scroll up is ^B, scroll down is ^F, etc.). Forward search, as you might expect, is initiated with /, and backward search is initiated with ?. Note that using ? for your first search from the bottom of a buffer as a forward search will find nothing, since you are at the bottom already. To get more complete `vi`-like instructions for `screen`, go to `screen`'s man page and search for "vi-like." See Figure 16-10.
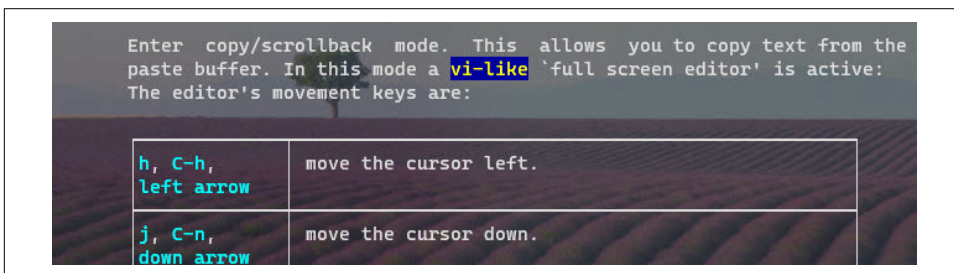
Figure 16-10. `vi` instructions for screen

With `screen` and its `vi`-like buffer navigation, you have access to information you'd probably thought you'd lost. That's been our experience more than once.

Take advantage of screen's key bindings

In the sample *.screenrc* code shown earlier, there are lines unrelated to defining terminal sessions. These are our recommended key bindings. For example, the lines:

```
# F2 : list windows
bindkey -k k2 windowlist
```

bind function key 2 (k2 denoting $\boxed{\text{F2}}$) to the command `windowlist`, which is normally invoked within a `screen` session by $\boxed{\text{CTRL-A}}$ $\boxed{\text{"}}$. This is a more intuitive shortcut for an action you'll probably use all the time: seeing the menu of available running `screen` sessions.

The others mapped in our sample configuration include:

$\boxed{\text{F3}}$

> Detach totally from the `screen` session.

$\boxed{\text{F10}}$

> Move to the `screen` session numerically minus one from the current session (e.g., if you are in session four, go to session three).

$\boxed{\text{F11}}$

> This is the complement to F10: go to the next available session.

$\boxed{\text{F12}}$

> Kill *all* sessions and quit `screen`. You probably won't use this often, but when you want to, it's nice to have a function key ready to use.

So now you know how to nicely map keys to common or interesting `screen` commands, but how do you keep track or remember what you mapped? For that, see the binding to $\boxed{\text{F9}}$:

```
# Help screen, key bindings:
bindkey -k k9 exec sed -n '/^# Help/s/^/^M/p;/^# F[1-9]/p' $HOME/.screenrc
```

We have bound $\boxed{\text{F9}}$ to execute the `sed` stream editor. It extracts the commented portion of all of the configuration file key mappings and displays that output at your prompt. Just like in the shell, everything including and beyond the # symbol is a comment. Typing $\boxed{\text{F9}}$ ends up looking something like this:

```
$                              F9 pressed
# Help screen, key bindings
# F2 : list windows
# F3 : detach screen (retains active sessions -- can reconnect later)
# F10: previous window (e.g., window 4 -> window 3)
# F11: next window (e.g., window 2 -> window 3)
# F12: kill all windows and quit screen (you will be prompted)
```

We chose $\boxed{\text{F9}}$ because some terminal emulators reserve $\boxed{\text{F1}}$ for their own help.

What makes screen great

Earlier, we posed the question, "With terminal emulators available today with multiple sessions, usually implemented across tabs, what makes `screen` worthwhile?"

The answer is that *you can leave the sessions and return to them*. They are maintained and active. As was just explained, we defined $\boxed{\text{F3}}$ to detach from `screen`. Try it.

You are returned to the command prompt and shell you were in *before* you entered screen. Try F2 now and verify it does nothing since you are not part of a screen session.

You can discover which screen session(s) you can attach to with screen's "list" command:

```
$ screen -list
There is a screen on:
        8491.pts-0.office-win10 (04/06/21 18:58:39)      (Detached)
1 Socket in /run/screen/S-elhannah.
```

Reattach and resume any and all work in that set of screen sessions with the attach option. You attach by entering the ID number:

```
$ screen -Ar 8491
```

You are now back in your set of sessions within screen. No worries about closing the window that is running screen; that qualifies as a "detach," and you reattach as just described! We have used this feature and sustained multiple sessions within screen (typically between 5 and 8), detaching and reattaching from various remote logins, and kept those sessions active for more than three months at a time!

And …, Browsers!

In many ways, internet browsers have been playing catch-up with technology since the day they were introduced. When browsers first appeared, they were crude (by today's standards), displaying information with links to click on and other information. Graphics were primitive, printing didn't exist, and a hodgepodge of standards created an unpleasant and inconsistent user experience.

Thankfully, standards have matured and merged, graphics and presentation became powerful, compatible, and portable (mostly), and browsers today are consistent and usable by casual users. Recently, third parties have provided extensions that bring Vim abstractions to the browser experience.

We have two favorite Chrome extensions we'll demonstrate: Wasavi, an implementation of Vim for editing text fields in the browser (e.g., filling out a text area in a customer feedback form), and Vimium, a way to navigate pages, bookmarks, URLs, and searches using Vim-like abstractions. Both of these extensions can dramatically improve your browsing efficiency.

These examples are for Chrome, implying they are available for any Chromium-based browser. This is good news for Microsoft Edge users, as Microsoft provides Chrome extension compatibility with Edge.

Wasavi

Wasavi is an open source plug-in for Chrome. The source code is available on GitHub.

Let's take a look. Figure 16-11 presents a text widget with lines that need to be moved.
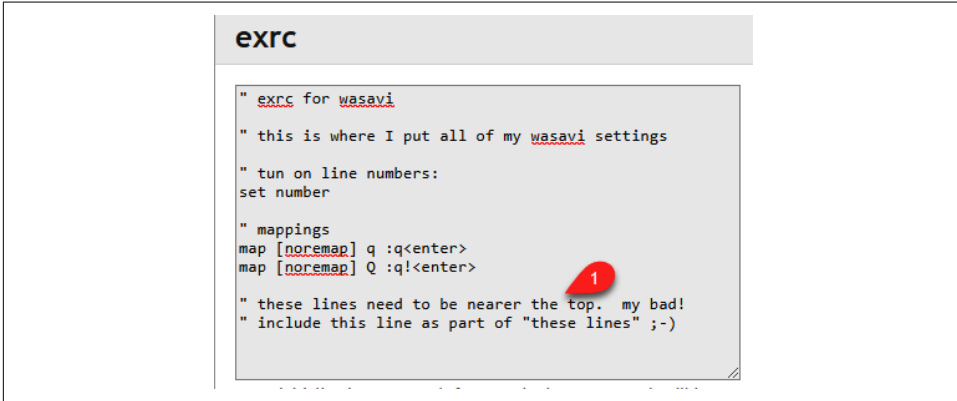
Figure 16-11. A browser text widget to be edited

1. Lines to be moved to a different location in a text widget. Let's do it the vi way with Wasavi.

Figure 16-12 presents the same text widget, but with Wasavi in action.
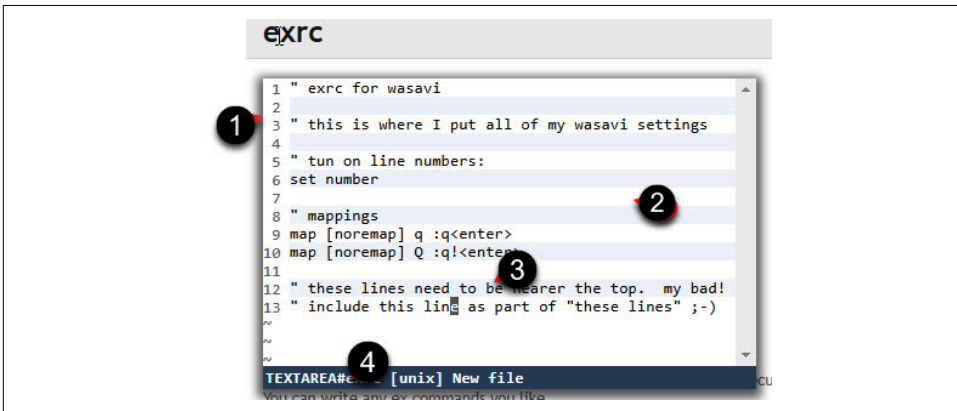
Figure 16-12. A browser text widget to be edited with Wasavi

1. Numbered lines give us a visual cue we are in vi.

2. The alternating background shade gives us another visual cue.

3. We can use `vi` commands to move these lines.

4. Wasavi's `vi` status line!

Finally, Figure 16-13 shows the same text widget after we've rearranged the contents.
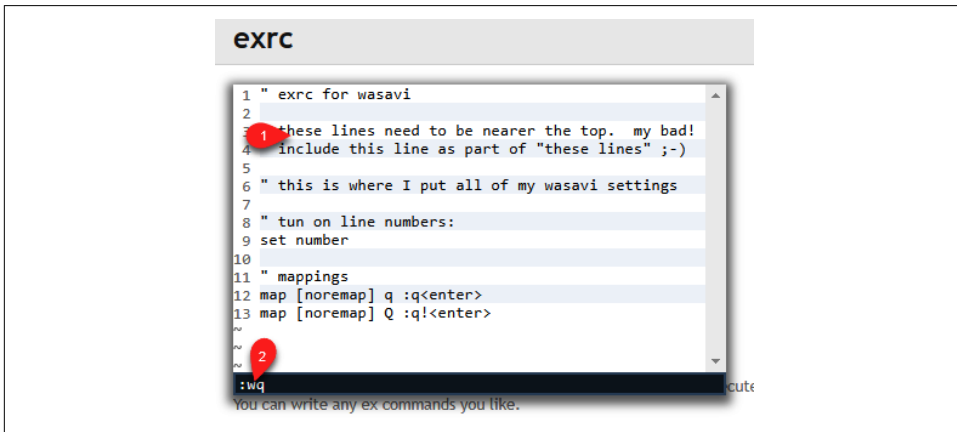
Figure 16-13. The text widget, after editing

1. The simple `vi` steps `11G`, `3dd`, `gg`, `p` do the trick. (`11G` goes to line 11; `3dd` deletes the three lines that we want to move; `gg` goes to the top of the file; and `p` puts the three lines we deleted into their new location.)

2. Saving the "file."

Vim + Chromium = Vimium

Vimium is a Chrome extension providing a Vim-like experience. You can read about it at its home page. It's also available in some form for other browsers, e.g., Firefox and Safari, for example.

Vimium lets you navigate the internet in a `vi`-like way. Since a browser is *not* an editor, of course, it makes sense that there is not a one-to-one mapping of Vim commands to Vimium actions. However, if you are familiar with and comfortable in Vim, the learning curve is pretty easy, and the Vim-isms make sense (mostly). Very much in the flavor of Vim, Vimium transforms the browser experience from a point-and-click exercise to the familiar "do everything from the keyboard" one. We find Vimium to be a must for browsing.

Keep the Vimium control handy

We recommend pinning the Vimium extension to the Chrome list of extensions; that is, keep it visible. This provides a visual cue indicating whether Vimium is *active* or

inactive. It also serves as a toggle should you need to temporarily turn Vimium off for bad behavior, which happens occasionally. It's rare that you would toggle Vimium on; for most websites where Vimium is off, it's off for a reason—on Google Mail, for example, where Google has already provided a full suite of key mappings to navigate mail.

There are so many things Vimium does. We'll highlight some useful features to get you going.

Finding links, and going to a link without clicking

Vimium abstracts the Vim f command by highlighting all visible links in the browser with *badges*. Typically these badges take the form of unique one- or two-character IDs. This powerful feature lets you quickly go to links, bypassing the need to slide the mouse and (accurately) click a link. For pages with minimal links, this usually takes only two keystrokes. The first keystroke is always f, and the second is the one- or two-character ID of the link. For a small number of links, Vimium tries to minimize IDs to a single character.

For web pages with many links, in addition to making it quick and easy to access a link, Vimium provides a uniform presentation of IDs and makes it simple to see what links exist on the page. Consider the sample Facebook page in Figure 16-14, with numbered explanations of the sections of the page. Depending on the resolution of your media, the badges/IDs may not be legible—they are clear when browsing.

1. SC goes to the author's Facebook profile.
2. AD goes to the author's friends list.
3. FD goes to the author's groups list.
4. DE goes to the shortcut to "The Vim text editor" Facebook forum.
5. XX, where *XX* is any matching ID to contact, initiates a message with that contact.
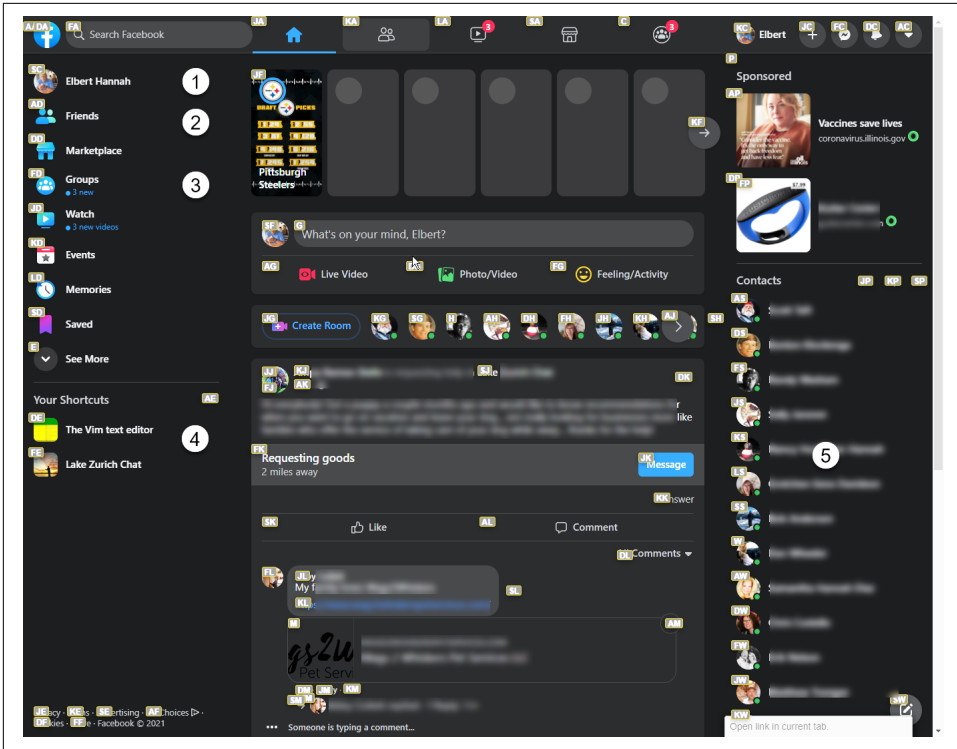
Figure 16-14. A Facebook page with Vimium badges

Vimium abstracts the Vim F command similarly to what it does for f, but going one step further by opening the link associated with the ID in a new tab.

Text search

Just as / initiates a search in Vim, Vimium initiates a search the same way. It's important to verify that the search is initiated by looking for the input box at the bottom of the browser window. See Figure 16-15.

Figure 16-15. Searching for text with Vimium

Most savvy browser users already know that CTRL-F or CMD-F for Mac invokes the native browser search, but in the spirit of a Vim experience, / is a more natural and convenient "stay on the home keys" experience. As you type in a search string, the browser scrolls to the first match and highlights the matching pattern. Hitting ENTER terminates the search string.

Just as `n` and `N` go to the next and previous matches of a search pattern in Vim, Vimium positions the browser to next and previous matches the same way.

> Vimium defaults to plain text searches, i.e., what you see is what you find. It does support regular expressions, in options that we leave for you to explore.

Browser navigation

Vimium abstracts Vim movements for navigating browser history and browser tabs. Here is a brief summary of what should be familiar to you:

`j`, `k`
> Scroll the browser up and down by lines at a time.

`H`, `L`
> Go to the "back" and "forward" pages, respectively. Note the case—it is case sensitive!

`K`, `J`
> Go to the tab immediately to the right and left, respectively. Again, case sensitive!

We use these two familiar Vim commands to go to the top or bottom of the web page:

`gg`
> Go to the top of the web page.

`G`
> Go to the bottom of the web page.

Other commands help you move between pages and tabs:

`]]`, `[[`
> For websites that have multiple pages linked as a forward and backward series, go to the next or previous page. Typically such pages have left- and right-arrow buttons on the bottom, or "NEXT" and "PREV" buttons providing the links.
>
> A good example of this would be lists of product reviews on a shopping website that has many pages of reviews. Also, for sites with annoying clickbait slide shows, this mechanism is wonderful for cruising through the slides without having to fight off the "click mes."

`^` *(up arrow or caret)*
> Visit the last visited tab. For example, if you have many tabs up and are focusing on two that are geographically far away from each other, the `^` character quickly jumps back and forth between the two most recently used tabs.

Useful key remappings

We found the `K` (go to immediate right tab) and `J` (go to immediate left tab) commands to be confusingly contrary to the intuitive and matching sense of natural order. That is, since `k` is a "scroll up," it would seem more natural for `K` to move to the immediate left tab instead of the immediate right tab. Fortunately, this can be remapped in Vimium options, and that's what we did.

Similarly, in the section "Marking Your Place" on page 62, we described how the `vi` command `''` (two apostrophes) returns to the beginning of the line of the previous mark or context. We find it useful to map `''` to Vimium's `visitPreviousTab` command (the same as `^`, just described). This lets you jump back and forth between tabs by typing a single quote consecutively.

To do this, open the Vimium options and enter the custom key mappings, as shown in Figure 16-16. This includes all three key remappings, as well as a new mapping for `q`.
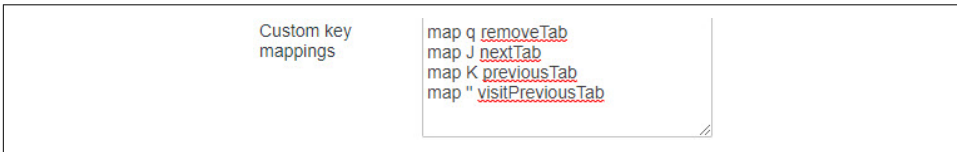
Figure 16-16. Useful Vimium key remappings

When you get lost and confused

In any web page where Vimium is active, you can quickly get help by typing ?. Vimium overlays the page with a summary of the commands and their actions. It is notable and convenient that Vimium actions work on Vimium's own help page. This comes in handy for searching for specific features. The ESC key returns you to your original web page. Figure 16-17 provides a partial snapshot of Vimium's help overlay.
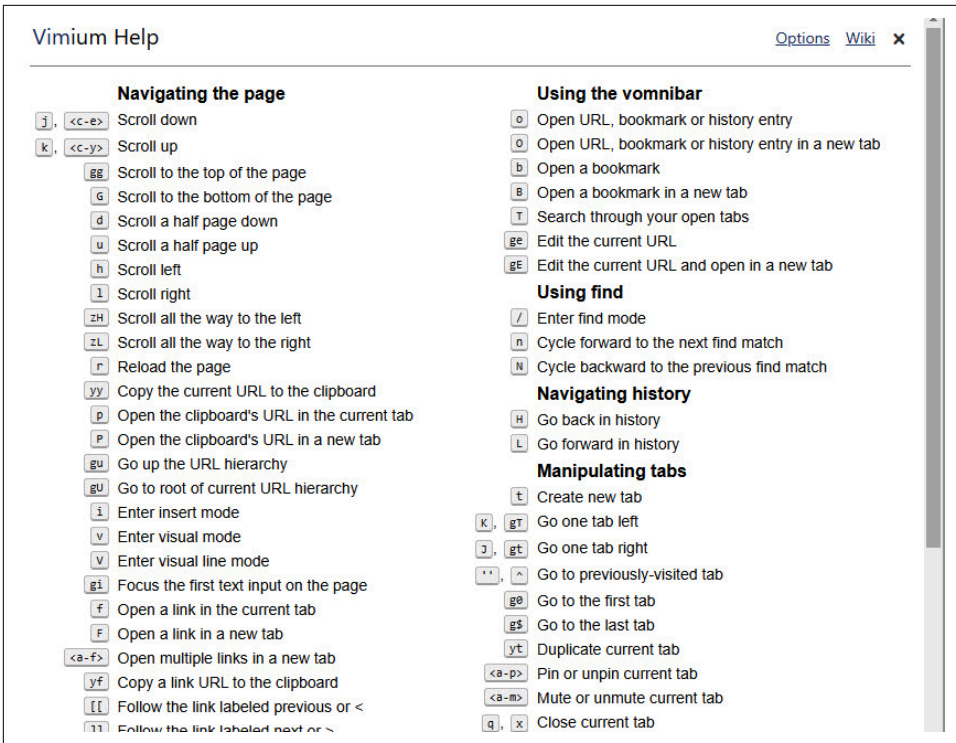
Figure 16-17. Vimium's help

A use case

Consider a common activity: shopping for a laptop online, such as on a large ecommerce website.

One important requirement is that the new laptop have excellent battery life. Once a candidate product is identified, it is easy to go and view all the reviews, knowing in advance that the reviews are sequenced across some number of "next" and "previous" pages. Starting on the first page, we search for any reference to *battery* with /battery ENTER. Now we can navigate up and down in the page of reviews to all the instances and references to reviewers' comments about battery life.

Next, since Vimium knows the common links that go to the next and previous sequenced pages, we use [[and]] to move forward and backward through the pages of reviews. Vimium remembers the search pattern, so it's easy to use n and N to find more comments in reviews related to *battery*. This is a common exercise that we use to quickly research products. Think about power, reliability, and many other desirable characteristics in a product.

vi for MS Word and Outlook

Many software developers who work in a Unix or GNU/Linux environment find themselves having to use Microsoft Word for documentation and Microsoft Outlook for email in their office environment.

As software developers, it is our shared experience that switching gears after long coding exercises using Vim is always a speed bump. Being able to use `vi` commands for editing in Word would allow developers like us to create better documentation faster. As it is, we end up getting in and out of Word as quickly as possible.

Similarly, we find Outlook to be something we have to fight with, instead of using it comfortably. A simple and common use case is a conversation thread in which it's important to reference comments in the thread. We would prefer to find relevant text and quote it by copying and pasting into our narrative. *If only* we could use `/` to search for text, `y` to yank it, and then `''` (return to last location) and `p`, the time savings would be enormous. And the efficiencies gained from not burning calories navigating, highlighting, and copying and pasting would improve thought processes. Instead of diverting to a mouse, scrolling, getting stuff, and pasting it, we could rely on `vi` muscle memory and quickly manipulate text more naturally, while focusing our important mental energy on what we are trying to say. This is aligned with the `vi` meme "editing at the speed of thought."

Fortunately there is a commercial plug-in for Word and Outlook that solves these problems: ViEmu.

Arnold no longer uses Word or Outlook. Elbert does. While he was skeptical at first, Elbert gave it a try and eventually ended up buying the plug-in. As Outlook's editing paradigm is essentially the same as that of MS Word, and since the plug-in is for both, we will simply talk about "Word," but be aware that everything applies to both Word and Outlook.

The behavior of ViEmu is very `vi`-like, and the plug-in is easily activated and deactivated to suit your needs. Since we have described the `vi` way many times by now, we won't dwell on how to use ViEmu. It is sufficient to know that there is enough `vi`-ness such that Vim users will instantly understand and appreciate how to use it.

Download the free trial from and install ViEmu. Once it is installed, you can verify that the plug-in is active by the yellow status line at the bottom of the application. See Figure 16-18.
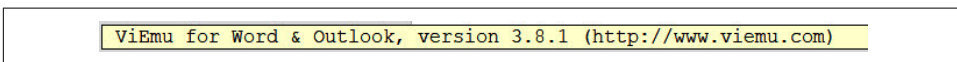
Figure 16-18. ViEmu's status line in Word or Outlook

Toggle ViEmu off by typing CTRL ALT SHIFT-V . Figure 16-19 shows the status line when ViEmu is inactive.

```
ViEmu disabled – use ViEmu settings or Ctrl-Shift-Alt-V to toggle it back on
```
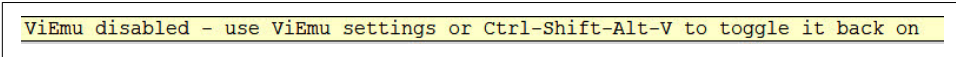
Figure 16-19. ViEmu's status line when ViEmu is inactive

Figure 16-20 shows the available options/preferences for the plug-in.
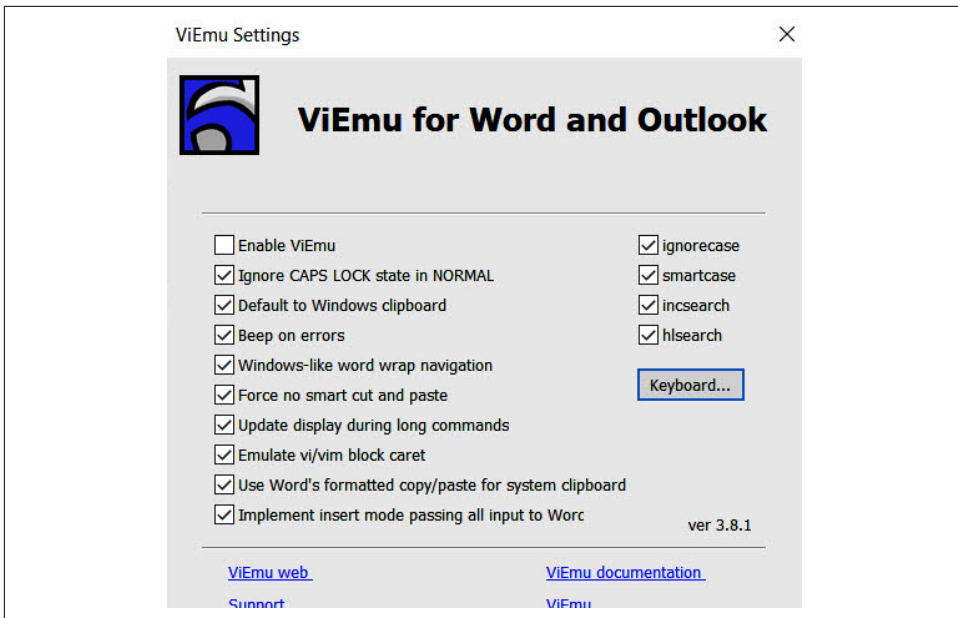
Figure 16-20. ViEmu settings

ViEmu strikes a balance between emulating vi functionality and staying faithful to Word. There are some things to note about the plug-in's behavior, some of which are the result of how each vi action or command is ultimately passed through to Word and handled by it. Consider the following points:

- Fonts are retained in the context of document.
- Ordered lists are automatically renumbered as they are shuffled with vi commands (e.g., delete a list item, and put it somewhere else in the list). We show an example shortly.
- Because Word has its own idea of what comprises lines in a document, the plug-in treats blocks of text in a kind of duality. For normal navigation up and down, the cursor moves intuitively up and down one line in the screen, much like it does with the arrow keys. However, since Word treats paragraphs as blocks,

the plug-in deletes an entire paragraph when you use the delete-line command, `dd`. This way of working is immediately recognizable.

Here is an example of the ordered list magic, using the first three habits from Stephen R. Covey's *The 7 Habits of Highly Effective People* (Simon & Schuster). While entering the list (see Figure 16-21), we realized that the order of the habits was wrong. Note that the cursor is on habit 2.

1. Begin With the End in Mind
2. Put First Things First ←
3. Be Proactive
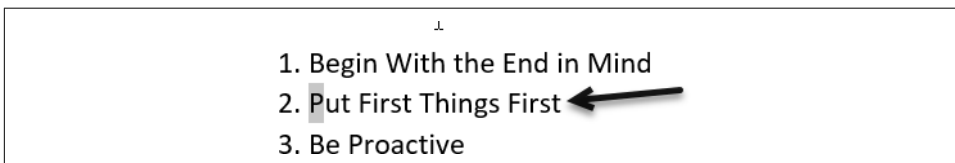
Figure 16-21. The first three of the "7 Habits," out of order

We type the `vi` command to swap lines, `ddp`, and voilà! They are now ordered properly. See Figure 16-22.

1. Begin With the End in Mind
2. Be Proactive
3. Put First Things First
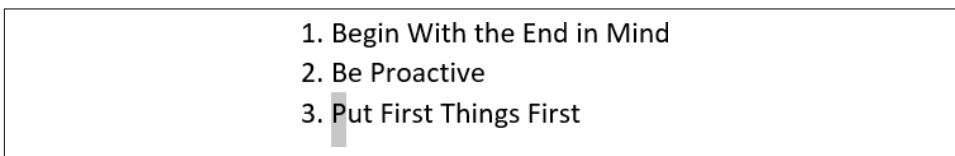
Figure 16-22. The first three habits, now in correct order

Again, all of these behaviors become comfortable and familiar. As an appetizer and assurance of ViEmu robustness, here is a partial list of observed and verified features:

- Change/delete in objects:
    — In parentheses: `ci(`, `di(`
    — In square brackets: `ci[`, `di[`
    — In curly braces: `ci{`, `di{`
- Delete at objects (deletion *including* the enclosing characters):
    — At parentheses: `da(`
    — At square brackets: `da[`
    — At curly braces: `da{`
- Full and selectable regular expression support. You can set the plug-in to honor your favorite regular expression flavor.
- Full documentation at *http://www.viemu.com/wo/viemu_doc.html*—always a big plus!

The rest of *how* to use ViEmu is left for you to explore. It's worth a look if you use Word a lot and prefer Vim behavior.

The plug-in website describes available options and pricing.

Honorable Mention: Tools with Some vi Features

It's difficult to define what *is* and what *isn't* vi. Take the case of regular expressions, which existed for many years preceding vi and Vim. However, some tools have some convenience features baked in that are worth mentioning because they have some flavor of what vi is all about: powerful fast editing, powerful search and replacement, and so on. We mention them here to highlight that tools with such features are out there and are worth looking for.

Google Mail

Google Mail has its own set of mapped keyboard shortcuts. You'll find some of the navigation mappings quite familiar. For most interactions, j and k are sufficient for navigating the list of emails. s "selects" the current line for any actions. x deletes any emails selected by s. mU marks selected emails as unread. ENTER opens the current email.

In an open email, x deletes the email, and u moves you "up" to the list of emails.

For a comprehensive list of keyboard shortcuts, enter ? in the listing panel (not in input mode).

Microsoft PowerToys

Microsoft has an add-on, PowerToys, with a myriad of cool additions to improve Windows. Most of them are outside the scope of this book. However, one tool worth mentioning is PowerToys' PowerRename utility. It provides a robust way to rename many files at once using criteria to accomplish what normally would be long and tedious one-at-a-time exercises. What sets PowerRename apart from similar tools is the ability to use regular expressions both to identify files to be renamed and to determine how the renaming is to be applied.

PowerToys also has a powerful keyboard mapping tool, Keyboard Manager, which lets you map the keyboard to your liking. While navigating a Windows desktop is unlikely to ever feel like a vi experience, at least there is now a way to easily map actions to vi-like keystrokes. Other, previous keyboard managers were either not free, not easy to manage, or both.

Summary

From well-known examples like command-line history editing, to software ecosystems like IDEs, to internet browsers, it's clear that `vi` is a popular editing paradigm. Anecdotally, Jon, the author of the ViEmu plug-in for Microsoft Word, received requests from Microsoft engineers for his plug-in! We have found that almost all of the highest-rated and most popular text editors either include a Vim setting for emulation or have plug-ins to emulate Vim behavior. Even Emacs has Viper, an installable plug-in for `vi` emulation! (See the section "Tastes Great, Less Filling" on page 480.) So if you ever sit down to a new environment, look around or ask others where the `vi`/Vim setting or plug-in is. It probably exists.

CHAPTER 17

Epilogue

If you've made it to here, we thank you for your patience and interest.

We've come a long way since the beginning of this book. We started with the basics—"What are text editors and text editing?"—and progressed through understanding `vi` command mode and the underlying `ex` editor, including regular expressions and the powerful command language underlying `vi` and Vim.

From there we did a deep dive into just some of Vim's many features, which make Vim at least an order of magnitude more powerful than the original `vi`. Vim is great at editing regular text (we composed this book in it), but it really shines as a programmer's editor.

Finally, we looked at how Vim's extensibility allows you to build it into a full-fledged IDE, as well as at how `vi`'s command-driven editing model can be found in other tools.

If and as long as you manage textual information from a keyboard, we think it well worth your time to learn Vim. Much as touch-typing is more efficient than hunt-and-peck typing, Vim is a quantum level better at text and program editing than any mouse-driven GUI editor.

Enjoy!

Appendixes

Part IV provides reference material that should be of interest to a `vi` or Vim user. This part contains the appendixes:

The vi, ex, and Vim Editors

This appendix summarizes the standard features of `vi` in quick-reference format. Commands entered at the colon prompt (known as `ex` commands because they date back to the original creation of that editor) are included, as well as the most popular Vim features.

This appendix presents the following topics:

- Command-line syntax
- Review of `vi` operations
- `vi` commands
- `vi` configuration
- `ex` basics
- Alphabetical summary of `ex` commands

Command-Line Syntax

The three most common ways of starting a Vim session are:

```
vim [options] file
vim [options] -c num file
vim [options] -c /pattern file
```

You can open *file* for editing, optionally at line *num* or at the first line matching *pattern*. If no *file* is specified, the editor opens with an empty buffer.

Command-Line Options

Because vi and ex are the same program, they share the same options. However, some options make sense for only one version of the program. Brackets indicate optional items. Command-line options specific to Vim are so marked:

+[`num`]
> Start editing at line number *num*, or at the last line of the file if *num* is omitted.

+/`pattern`
> Start editing at the first line matching *pattern*.[1]

+?`pattern`
> Start editing at the last line matching *pattern*.[1]

-b
> Edit the file in binary mode. {Vim}

-c `command`
> Run the given ex command upon startup. Only one -c option is permitted for vi; Vim accepts up to 10. The older form of this option, +*command*, is still supported.

--cmd `command`
> Like -c, but execute the *command* before any configuration files are read. {Vim}

-C
> Solaris 10 vi: same as -x, but assume the file is encrypted already. Not in Solaris 11 vi.
>
> Vim: start the editor in vi-compatible mode.

-d
> Run in diff mode. Works like vimdiff. {Vim}

-D
> Debugging mode for use with scripts. {Vim}

-e
> Run as ex (line-editing rather than full-screen mode).

-h
> Print help message, then exit. {Vim}

---

1 If you use Vim and set *.viminfo* to restore the file's last cursor location, the search starts in the direction mentioned *from* the restored cursor location. Also, depending on the setting of the wrapscan (ws) option in the *.vimrc* configuration file, the search will stop (:set nowrapscan) or continue (:set wrapscan) past the top or bottom line in the file.

-i *file*

    Use the specified *file* instead of the default (*~/.viminfo*) to save or restore Vim's state. {Vim}

-l

    Enter Lisp mode for running Lisp programs (not supported in all versions).

-L

    List files that were saved due to an aborted editor session or system crash (not supported in all versions). For Vim, this option is the same as `-r`.

-m

    Start the editor with the `write` option turned off so that the user cannot write to files. Vim still permits changes to the buffer but will not permit writing the file, even with the `:w!` override. {Vim}

-M

    Do not allow text in files to be modified. This is similar to `-m` but additionally blocks any changes to the buffer. {Vim}

-n

    Do not use a swap file; record changes in memory only. {Vim}

--noplugin

    Do not load any plug-ins. {Vim}

-N

    Run Vim in a non-`vi`-compatible mode. {Vim}

-o[*num*]

    Start Vim with *num* open windows. The default is to open one window for each file. {Vim}

-O[*num*]

    Start Vim with *num* open windows arranged horizontally (split vertically) on the screen. {Vim}

-r [*file*]

    Recovery mode; recover and resume editing on *file* after an aborted editor session or system crash. Without *file*, list files available for recovery.

-R

    Edit files in read-only mode. Vim warns you if you make changes and allows the changes to occur. If you try to save the file with changes, Vim prompts you to override the read-only context.

`-s`

Silent; do not display prompts. Useful when running a script. This behavior also can be set through the older - option. For Vim, applies only when used together with `-e`.

`-s` *scriptfile*

Read and execute commands given in the specified *scriptfile* as if they were typed in from the keyboard. {Vim}

`-S` *commandfile*

Read and execute commands given in *commandfile* after loading any files for editing specified on the command line. Shorthand for `vim -c 'source` *command file*`'`. {Vim}

`-t` *tag*

Edit the file containing *tag*, and position the cursor at its definition.

`-T` *type*

Set the `term` (terminal type) option. This value overrides the `$TERM` environment variable. {Vim}

`-u` *file*

Read configuration information from the specified configuration file instead of the default *.vimrc* configuration file. If the *file* argument is `NONE`, Vim reads no configuration files, loads no plug-ins, and runs in compatible mode. If the argument is `NORC`, it reads no configuration files, but it does load plug-ins. {Vim}

`-v`

Run in full-screen mode (default for `vi`).

`--version`

Print version information, then exit. {Vim}

`-V[`*num*`]`

Verbose mode; print messages about what options are being set and what files are being read or written. You can set a level of verbosity to increase or decrease the number of messages received. The default value is 10 for high verbosity. {Vim}

`-w` *rows*

Set the window size so *rows* lines at a time are displayed; useful when editing over a long-distance internet connection. Older versions of `vi` do not permit a space between the option and its argument. Vim does not support this option.

`-W` *scriptfile*

Write all typed commands from the current session to the specified *scriptfile*. The file thus created can be used with the `-s` option. {Vim}

-x

Prompt for a key that will be used to try to encrypt or decrypt a file using `crypt` (not supported in all versions).[2]

-y

Modeless `vi`; run Vim in insert mode only, without a command mode. This is the same as invoking Vim as `evim`. {Vim}

-Z

Start Vim in restricted mode. Do not allow shell commands or suspension of the editor. {Vim}

Although most people know `ex` commands only by their use within `vi`, the editor also exists as a separate program and can be invoked from the shell (for instance, to edit files as part of a script). Within `ex`, you can enter the `vi` or `visual` command to start `vi`. Similarly, within `vi`, you can enter `Q` to quit the `vi` editor and enter `ex`.

You can exit `ex` in several ways:

`:x`   Exit (save changes and quit).

`:q!`   Quit without saving changes.

`:vi`   Enter the `vi` editor.

Review of vi Operations

This section provides a review of the following:

- `vi` modes
- Syntax of `vi` commands
- Status-line commands

Command Mode

Once the file is opened, you are in command mode. From command mode, you can:

- Invoke insert mode
- Issue editing commands
- Move the cursor to a different position in the file
- Invoke `ex` commands

---

2 This option is obsolete; the `crypt` command's encryption is weak and you shouldn't use it.

- Invoke a shell
- Save the current version of the file
- Exit the editor

Insert Mode

In insert mode, you enter new text into the file. You normally enter insert mode with the i command. Press the ESC key to exit insert mode and return to command mode. The full list of commands that enter insert mode is provided later, in the section "Insert Commands" on page 425.

Syntax of vi Commands

In vi, editing commands have the following general form:

    [n] operator [m] motion

The basic editing *operators* are:

c   Begin a change.

d   Begin a deletion.

y   Begin a yank (or copy).

If the current line is the object of the operation, the *motion* is the same as the operator: cc, dd, yy. Otherwise, the editing operators act on objects specified by cursor-movement commands or pattern-matching commands. For example, cf. changes up to the next period. *n* and *m* are the number of times the operation is performed, or the number of objects the operation is performed on. If both *n* and *m* are specified, the effect is $n \times m$.

An object of operation can be any of the following text blocks:

word
    Includes characters up to a whitespace character (space or tab) or punctuation mark. A capitalized object is a variant form that recognizes only whitespace.

sentence
    Up to ., !, or ?, followed by two spaces. Vim looks only for a single following space.

paragraph
    Up to the next blank line or nroff/troff paragraph macro defined by the para= option.

section
> Up to the next `nroff`/`troff` section heading defined by the `sect=` option.

motion
> Up to the character or other text object as specified by a motion specifier, including pattern searches.

Examples

`2cw`	Change the next two words.
`d}`	Delete up to the next paragraph.
`d^`	Delete back to the beginning of the line.
`5yy`	Copy the next five lines.
`y]]`	Copy up to the next section.
`cG`	Change to the end of the edit buffer.

More commands and examples may be found later in this appendix, in the section "Changing and deleting text" on page 426.

Visual mode (Vim only)

Vim provides an additional facility, *visual mode*. This allows you to highlight blocks of text, which then become the object of edit commands such as deletion or saving (yanking). Graphical versions of Vim allow you to use the mouse to highlight text in a similar fashion. See the earlier section "Visual Mode Motion" on page 175 for more information.

`v`	Select text in visual mode one character at a time.
`V`	Select text in visual mode one line at a time.
CTRL-V	Select text in visual mode in blocks.

Status-Line Commands

Most commands are not echoed on the screen as you input them. However, the status line at the bottom of the screen is used to edit these commands:

`/`	Search forward for a pattern.
`?`	Search backward for a pattern.
`:`	Invoke an `ex` command.
`!`	Invoke a Unix command that takes as its input an object in the buffer and replaces it with output from the command. You type a motion command after the `!` to describe what should be passed to the Unix command. The command itself is shown on the status line.

Commands that are entered on the status line must be completed by pressing the ENTER key.[3] In addition, error messages and output from the CTRL-G command are displayed on the status line.

vi Commands

vi supplies a large set of single-key commands when in command mode. Vim supplies additional multikey commands.

Movement Commands

Some versions of vi do not recognize extended keyboard keys (e.g., arrow keys, page up, page down, home, insert, and delete); some do. All versions, however, recognize the keys in this section. Many users of vi prefer to use these keys, as it helps them keep their fingers on the home row of the keyboard. A number preceding a command repeats the movement. Movement commands are also used after an operator. The operator works on the text that is moved.

Character

h, j, k, l	Left, down, up, right (← , ↓ , ↑ , →)
Space bar	Right
BACKSPACE	Left
CTRL-H	Left
CTRL-N	Down
CTRL-P	Up

Text

w, b	Forward, backward by "word" (letters, numbers, and underscores make up words).
W, B	Forward, backward by "Word" (only whitespace separates items).
e	End of word.
E	End of Word.
ge	End of previous word. {Vim}
gE	End of previous Word. {Vim}
), (	Beginning of next, current sentence.

---

3 If you are using the original vi or have set the Vim compatible option, the ESC key will execute the command. This can be unexpected. Vim (in nocompatible mode) simply cancels the command and takes no action.

}, {	Beginning of next, current paragraph.
]], [[	Beginning of next, current section.
][, [ ]	End of next, current section. {Vim}

Lines

Long lines in a file may show up on the screen as multiple lines. They wrap around from one screen line to the next. Although most commands work on the lines as defined in the file, a few commands work on lines as they appear on the screen. The Vim option `wrap` allows you to control how long lines are displayed.

0, $	First, last position of the current line.
^, _	First nonblank character of the current line.
+, -	First nonblank character of the next, previous line.
[ENTER]	First nonblank character of the next line.
num |	Column *num* of the current line.
g0, g$	First, last position of the screen line. {Vim}
g^	First nonblank character of the screen line. {Vim}
gm	Middle of the screen line. {Vim}
gk, gj	Move up, down one screen line.[a] {Vim}
H	Top line of the screen (Home position).
M	Middle line of the screen.
L	Last line of the screen.
num H	*num* lines after the top line.
num L	*num* lines before the last line.

[a] These are redundant, since removing the g results in the same action.

Screens

[CTRL-F], [CTRL-B]	Scroll forward, backward one screen.
[CTRL-D], [CTRL-U]	Scroll down, up a half screen.
[CTRL-E], [CTRL-Y]	Show one more line at the bottom, top of the screen. If possible, Vim maintains the cursor position on the same line—e.g., if the cursor is on line 50 and you use this movement, the cursor moves up or down to stay on line 50.
z [ENTER]	Reposition the line with the cursor to the top of the screen.
z.	Reposition the line with the cursor to the middle of the screen.
z-	Reposition the line with the cursor to the bottom of the screen.
[CTRL-L]	Redraw the screen (without scrolling).
[CTRL-R]	`vi`: redraw the screen (without scrolling).
	Vim: redo the last undone change.

Within a Screen

H   Move to home—the first character of the top line on the screen.

M   Move to the first character of the middle line on the screen.

L   Move to the first character of the last line on the screen.

n H   Move to the first character of the line *n* lines below the top line.

n L   Move to the first character of the line *n* lines above the last line.

Searches

/*pattern*	Search forward for *pattern*. End with ENTER.
/*pattern*/+ *num*	Go to line *num* after *pattern*. Forward search for *pattern*.
/*pattern*/- *num*	Go to line *num* before *pattern*. Forward search for *pattern*.
?*pattern*	Search backward for *pattern*. End with ENTER.
?*pattern*?+ *num*	Go to line *num* after *pattern*. Backward search for *pattern*.
?*pattern*?- *num*	Go to line *num* before *pattern*. Backward search for *pattern*.
:noh	Suspend search highlighting until next search. {Vim}
n	Repeat the previous search.
N	Repeat the previous search in the opposite direction.
/	Repeat the previous search forward. End with ENTER.
?	Repeat the previous search backward. End with ENTER.
*	Search forward for the word under the cursor. Matches only exact words. {Vim}
#	Search backward for the word under the cursor. Matches only exact words. {Vim}
g*	Search forward for the word under the cursor. Matches the characters of this word when embedded in a longer word. {Vim}
g#	Search backward for the word under the cursor. Matches the characters of this word when embedded in a longer word. {Vim}
%	Find match of current parenthesis, brace, or bracket.
f *x*	Move the cursor forward to *x* on current line.
F *x*	Move the cursor backward to *x* on current line.
t *x*	Move the cursor forward to the character before *x* in current line.
T *x*	Move the cursor backward to the character after *x* in current line.
,	Reverse the search direction of the last f, F, t, or T.
;	Repeat the last f, F, t, or T.

Line numbering

CTRL-G   Display the current line number.

gg   Move to the first line in the file. {Vim}

---

num G	Move to line number *num*.
G	Move to the last line in the file.
: *num*	Move to line number *num*.

Marks

m *x*	Place mark *x* at the current position.
` *x*	(Backquote.) Move cursor to mark *x*.
' *x*	(Apostrophe.) Move to the start of the line containing *x*.
``	(Backquotes.) Return to the position before the most recent jump.
''	(Apostrophes.) Like preceding, but return to the start of the line.
'"	(Apostrophe quote.) Move to the position when last editing the file. {Vim}
`[, `]	(Backquote bracket.) Move to the beginning/end of the previous text operation. {Vim}
'[, ']	(Apostrophe bracket.) Like preceding, but return to the start of the line where the operation occurred. {Vim}
`.	(Backquote period.) Move to the last change in the file. {Vim}
'.	(Apostrophe period.) Like preceding, but return to the start of the line. {Vim}
'0	(Apostrophe zero.) Move to where you were when you last exited Vim. {Vim}
:marks	List active marks. {Vim}

Insert Commands

a	Append after the cursor.
A	Append to the end of the line.
c	Begin change operation.
C	Change to the end of the line.
gi	Insert at the place at which you were last editing the file. {Vim}
gI	Insert at the beginning of the line. {Vim}
i	Insert before the cursor.
I	Insert at the beginning of the line.
o	Open a line below the cursor.
O	Open a line above the cursor.
R	Begin overwriting text.
s	Substitute a character.
S	Substitute the entire line.
ESC	Terminate insert mode.

The following commands work in insert mode:

BACKSPACE	Delete the previous character.
DELETE	Delete the current character.
TAB	Insert a tab.
CTRL-A	Repeat the last insertion. {Vim}
CTRL-D	Shift line left to the previous shiftwidth.
^ CTRL-D	Shift cursor to the beginning of the line, but only for one line.
0 CTRL-D	Shift cursor to the beginning of the line, and reset the autoindent level to zero.
CTRL-E	Insert the character found just below the cursor. {Vim}
CTRL-H	Delete the previous character (same as backspace).
CTRL-I	Insert a tab.
CTRL-K	Begin insertion of a multikeystroke character. {Vim}
CTRL-N	Insert the next completion of the pattern to the left of the cursor. {Vim}
CTRL-P	Insert the previous completion of the pattern to the left of the cursor. {Vim}
CTRL-T	Shift the line right to the next shiftwidth. {Vim}
CTRL-U	Delete the current line.
CTRL-V	Insert the next character verbatim.
CTRL-W	Delete the previous word.
CTRL-Y	Insert the character found just above the cursor. {Vim}
CTRL-[	(ESC) Terminate insert mode.

Some of the control characters listed in the previous table are set by `stty`. Your terminal emulator's settings may differ.

Edit Commands

Recall that `c`, `d`, and `y` are the basic editing operators.

Changing and deleting text

The following list is not exhaustive, but it illustrates the most common operations:

cw	Change a word.
cc	Change the line.
c$	Change text from the current position to the end of the line.
C	Same as c$.
dd	Delete the current line.
num dd	Delete *num* lines.
d$	Delete text from the current position to the end of the line.

D	Same as d$.
dw	Delete a word.
d}	Delete up to the next paragraph.
d^	Delete back to the beginning of the line.
d/*pattern*	Delete up to the first occurrence of *pattern*.
dn	Delete up to the next occurrence of *pattern*.
df *x*	Delete up to and including *x* on the current line.
dt *x*	Delete up to (but not including) *x* on the current line.
dL	Delete up to the last line on the screen.
dG	Delete to the end of the file.
gqap	Reformat the current paragraph to `textwidth`. {Vim}
g~w	Switch the case of word. {Vim}
guw	Change word to lowercase. {Vim}
gUw	Change word to uppercase. {Vim}
p	Insert the last deleted or yanked text after the cursor.
P	Insert the last deleted or yanked text before the cursor.
gp	Same as p, but leave the cursor at the end of the inserted text. {Vim}
gP	Same as P, but leave the cursor at the end of the inserted text. {Vim}
]p	Same as p, but match the current indentation. {Vim}
[p	Same as P, but match the current indentation. {Vim}
r *x*	Replace character with *x*.
R *text*	Replace with new *text* (overwrite), beginning at cursor. ESC ends replace mode.
s	Substitute one character.
4s	Substitute four characters.
S	Substitute the entire line.
u	Undo the last change.
CTRL-R	Redo the last change. {Vim}
U	Restore the current line.
x	Delete the character at the current cursor position.
X	Delete back one character.
5X	Delete the previous five characters.
.	Repeat the last change.
~	Reverse case and move the cursor right. {vi, and Vim with option `notildeop`}
~w	Reverse the case of a word. {Vim with option `tildeop`}
~~	Reverse the case of the line. {Vim with option `tildeop`}
CTRL-A	Increment the number under the cursor. {Vim}
CTRL-X	Decrement the number under the cursor. {Vim}

Copying and moving

Register names are the letters `a–z`. Uppercase names append text to the corresponding register:

Y	Copy the current line.
yy	Copy the current line.
" *x* yy	Copy the current line to register *x*.
ye	Copy text to the end of the word.
yw	Like `ye`, but include the whitespace after the word.
y$	Copy the rest of the line.
" *x* dd	Delete the current line into register *x*.
" *x* d *motion*	Delete into register *x*.
" *x* p	Put contents of register *x*.
y]]	Copy up to the next section heading.
J	Join the current line to the next line.
gJ	Same as `J`, but without inserting a space. {Vim}
:j	Same as `J`.
:j!	Same as `gJ`.

Saving and Exiting

Writing a file means overwriting the file with the current text in the editing buffer.

ZZ	Quit `vi`, writing the file only if changes were made.
:x	Same as `ZZ`.
:wq	Write the file and quit.
:w	Write the file.
:w *file*	Save a copy to *file*.
:*n,m*w *file*	Write lines *n* to *m* to new *file*.
:*n,m*w >> *file*	Append lines *n* to *m* to existing *file*.
:w!	Write the file (overriding protection).
:w! *file*	Overwrite *file* with the current text.
:w %.new	Write the current buffer named *file* as *file*.new.
:q	Quit the editor (fails if changes were made).
:q!	Quit the editor (discarding edits).
Q	Quit `vi` and invoke `ex`.
:vi	Return to `vi` after `Q` command.
%	Replaced with the current filename in editing commands.

#	Replaced with the alternate filename in editing commands.

Accessing Multiple Files

:e *file*	Edit another *file*; the current file becomes the alternate, named by #.
:e!	Return to the version of the current file as of the time of the last write.
:e + *file*	Begin editing at the end of *file*.
:e +*num* *file*	Open *file* at line *num*.
:e #	Open to the previous position in the alternate file.
:ta *tag*	Edit the file at location *tag*.
:n	Edit the next file in the list of files.
:n!	Force editing of the next file.
:n *files*	Specify a new list of *files*.
:rewind	Edit the first file in the argument list.
CTRL-G	Show the current file and line number.
:args	Display the list of files being edited.
:prev	Edit the previous file in the list of files. {Vim}
:last	Edit the last file in the list of files. {Vim}

Window Commands (Vim)

The following table lists common commands for controlling windows in Vim. See also the split, vsplit, and resize commands in the section "Alphabetical Summary of ex Commands" on page 435. For brevity, control characters are marked in the following list by ^.

> All single-character keys are lowercase. Uppercase keys are denoted with a "SHIFT-" prefix.

:new	Open a new window.
:new *file*	Open *file* in a new window.
:sp[lit] [*file*]	Split the current window. With *file*, edit that file in the new window.
:sv[split] [*file*]	Same as :sp, but make the new window read-only.
:sn[ext] [*file*]	Edit the next file in the file list in the new window.
:vsp[lit] [*file*]	Like :sp, but split vertically instead of horizontally.
:clo[se]	Close the current window.
:hid[e]	Hide the current window, unless it is the only visible window.

`:on[ly]`	Make the current window the only visible one.
`:res[ize]` *num*	Resize the window to *num* lines.
`:wa[ll]`	Write all changed buffers to their files.
`:qa[ll]`	Close all buffers and exit.
`CTRL-W` `S`	Same as `:sp`.
`CTRL-W` `N`	Same as `:new`.
`CTRL-W` `^`	Open the new window with the alternate (previously edited) file.
`CTRL-W` `C`	Same as `:clo`.
`CTRL-W` `O`	Same as `:only`.
`CTRL-W` `J`	Move the cursor to the next window.
`CTRL-W` `K`	Move the cursor to the previous window.
`CTRL-W` `P`	Move the cursor to the previous window.
`CTRL-W` `H` / `CTRL-W` `L`	Move the cursor to the window on the left/right of screen.
`CTRL-W` `T` / `CTRL-W` `B`	Move the cursor to the window on the top/bottom of screen.
`CTRL-W` `SHIFT-K` / `CTRL-W` `SHIFT-B`	Move the current window to the top/bottom of screen.
`CTRL-W` `SHIFT-H` / `CTRL-W` `SHIFT-L`	Move the current window to the far left/right of screen.
`CTRL-W` `R` / `CTRL-W`, `SHIFT-R`	Rotate the windows down/up.
`CTRL-W` `+` / `CTRL-W` `-`	Increase/decrease the current window size.
`CTRL-W` `=`	Make all windows the same height.

Interacting with the System

`:r` *file*	Read in the contents of *file* after the cursor.
`:r !`*command*	Read in the output from *command* after the current line.
`:`*num*`r !`*command*	Like previous, but place after line *num* (use zero for the top of the file).
`:!` *command*	Run *command*, then return.
`!`*motion command*	Send the text covered by *motion* to *command*; replace with output.
`:`*n,m*`!`*command*	Send lines *n–m* to *command*; replace with output.
num`!!` *command*	Send *num* lines to *command*; replace with output.
`:!!`	Repeat the last system command.
`:sh`	Create a subshell; return to editor with *EOF*.
`CTRL-Z`	Suspend the editor, resume with `fg`. This iconifies `gvim`.
`:so` *file*	Read and execute ex commands from *file*.

Macros

`:ab` *in out*	Use *in* as an abbreviation for *out* in insert mode.
`:unab` *in*	Remove the abbreviation for *in*.

`:ab`	List abbreviations.
`:map` *`string sequence`*	Map characters *string* as *sequence* of commands. Use #1, #2, etc. for the function keys.
`:unmap` *`string`*	Remove the map for characters *string*.
`:map`	List character strings that are mapped.
`:map!` *`string sequence`*	Map characters *string* to input mode *sequence*.
`:unmap!` *`string`*	Remove input mode map for characters *string* (you may need to quote the characters with CTRL-V).
`:map!`	List character strings that are mapped for input mode.
`q` *`x`*	Record typed characters into the register specified by letter *x*. If the letter is uppercase, append to the register. {Vim}
`q`	Stop recording. {Vim}
`@`*`x`*	Execute the register specified by the letter *x*.
`@@`	Execute the last register command.

In `vi`, the following characters are unused in command mode and can be mapped as user-defined commands:

Letters
> g, K, q, V, and v

Control keys
> CTRL-A, CTRL-K, CTRL-O, CTRL-W, CTRL-X, CTRL-_, and CTRL-\

Symbols
> _, *, \, =, and #

> The = is used by `vi` if Lisp mode is set. Different versions of `vi` may use some of these characters, so test them before using.

Vim does not use CTRL-K, CTRL-_, or CTRL-\. See `:help noremap` in Vim for additional mapping functionality.

Miscellaneous Commands

`<`	Shift the text described by the following motion command left by one shiftwidth. {Vim}
`>`	Shift the text described by the following motion command right by one shiftwidth. {Vim}
`<<`	Shift the line left by one shiftwidth (default is eight spaces).
`>>`	Shift the line right by one shiftwidth (default is eight spaces).
`>}`	Shift right to the end of the paragraph.

`<%`	Shift left until the matching parenthesis, brace, or bracket. (The cursor must be on the matching symbol.)
`[count] ==`	Indent *count* lines in C-style, or using the program specified in the `equalprg` option. {Vim}
`g`	Start many multiple character commands in Vim.
`K`	Look up the word under the cursor in the manual pages (or via the program defined in `keywordprg`). {Vim}
CTRL-O	Return to the previous jump. {Vim}
CTRL-Q	Same as CTRL-V. {Vim} (On some terminals, resume data flow.)
CTRL-T	Return to the previous location in the tag stack. {Solaris `vi`, Vim.}
CTRL-]	Perform a tag lookup on the text under the cursor.
CTRL-\	Enter `ex` line-editing mode.
CTRL-^	Return to the previously edited file.

vi Configuration

This section describes the following:

- The `:set` command
- Example *.exrc* file

The :set Command

The `:set` command allows you to specify options that change characteristics of your editing environment. Options may be put in the *~/.exrc* or *~/.vimrc* files or set during an editing session.

The colon does not need to be typed if the command is put in *.exrc*:

`:set x`	Enable Boolean option *x*; show the value of other options.
`:set no x`	Disable option *x*. Do not supply a space between `no` and *x*.
`:set x = value`	Give *value* to option *x*.
`:set`	Show changed options.
`:set all`	Show all options.
`:set x ?`	Show the value of option *x*.

Appendix B, "Setting Options", provides tables of `:set` options for the "Heirloom" and Solaris versions of `vi`, and for Vim. Please see that appendix for more information.

Example .exrc File

In an `ex` script file, comments start with the double-quote character. The following lines of code are an example of a customized *.exrc* file:

```
set nowrapscan            " Searches don't wrap at end of file
set wrapmargin=7          " Wrap text at 7 columns from right margin
set sections=SeAhBhChDh nomesg  " Set troff macros, disallow message
map q :w^M:n^M            " Alias to move to next file
map v dwElp              " Move a word
ab ORA O'Reilly Media, Inc.  " Input shortcut
```

The `q` alias isn't needed for Vim, which has the `:wn` command. The `v` alias would hide the Vim command `v`, which enters character-at-a-time visual mode operation.

ex Basics

The `ex` line editor serves as the foundation for the screen editor `vi`. Commands in `ex` work on the current line or on a range of lines in a file. Most often, you use `ex` from within `vi`. In `vi`, `ex` commands are preceded by a colon and executed when you press ENTER.

You can also invoke `ex` on its own—from the command line—just as you would invoke `vi`. (You could execute an `ex` script this way.) Or you can use the `vi` command Q to enter `ex`.

Syntax of ex Commands

To enter an `ex` command from `vi`, type:

```
:[address] command [options]
```

An initial `:` indicates an `ex` command. As you type the command, it is echoed on the status line. Execute the command by pressing the ENTER key. *Address* is the line number or range of lines that are the object of *command*. *Options* and *addresses* are described later. `ex` commands are described in the section "Alphabetical Summary of ex Commands" on page 435.

You can exit `ex` in several ways:

`:x`	Exit (save changes and quit).
`:q!`	Quit without saving changes.
`:vi`	Switch to the `vi` editor on the current file.

Addresses

If no address is given, the current line is the object of the command. If the address specifies a range of lines, the format is:

    x,y

where *x* and *y* are the first and last addressed lines (*x* must precede *y* in the buffer). *x* and *y* each may be a line number or a symbol. Using `;` instead of `,` sets the current line to *x* before interpreting *y*. The notation `1,$` addresses all lines in the file, as does `%`.

Address Symbols

`1,$`	All the lines in the file.
x,y	Lines *x* through *y*.
x;y	Lines *x* through *y*, with the current line reset to *x* before computing *y*.
`0`	The top of the file.
`.`	The current line.
num	Absolute line number *num*.
`$`	The last line.
`%`	All lines; same as `1,$`.
x - n	*n* lines before *x*.
x + n	*n* lines after *x*.
`-[`*num*`]`	One or *num* lines previous.
`+[`*num*`]`	One or *num* lines ahead.
`'`*x*	(Apostrophe.) The line marked with *x*.
`''`	(Apostrophe apostrophe.) The previous mark.
`/`*pattern*`/`	Forward to the line matching *pattern*.
`?`*pattern*`?`	Backward to the line matching *pattern*.

See Chapter 6, "Global Replacement", for more information on using patterns.

Options

`!`

    Indicates a variant form of the command, overriding the normal behavior. The `!` must come immediately after the command.

count

    The number of times the command is to be repeated. Unlike in `vi` commands, *count* cannot precede the command, because a number preceding an

---

`ex` command is treated as a line address. For example, `d3` deletes three lines, beginning with the current line; `3d` deletes line 3.

file

The name of a file that is affected by the command. `%` stands for the current file; `#` stands for the previous file.

Alphabetical Summary of ex Commands

`ex` commands can be entered by specifying any unique abbreviation. In the following list of reference entries, the full name appears as the heading of the reference entry, and the shortest possible abbreviation is shown in the syntax line below it. Examples are assumed to be typed from `vi`, so they include the `:` prompt.

abbreviate

```
ab [string text]
```

Define *string* when typed to be translated into *text*. If *string* and *text* are not specified, list all current abbreviations.

Examples

Note: `^M` appears when you type `^V` followed by ENTER.

```
:ab ora O'Reilly Media, Inc.
:ab id Name:^MRank:^MPhone:
```

append

```
[address] a[!]
text
.
```

Append new *text* at the specified *address*, or at the present address if none is specified. Add a `!` to toggle the `autoindent` setting that is used during input. That is, if `autoindent` was enabled, `!` disables it. Enter new text after entering the command. Terminate the input of new text by entering a line consisting of just a period.

Example

```
:a                       Begin appending to the current line
Append this line
and this line too.
.                        Terminate input of text to append
```

args

```
ar
args file ...
```

Print the members of the argument list (files named on the command line), with the current argument (the file being edited) printed in brackets ([ ]).

The second syntax is for Vim, which allows you to reset the list of files to be edited.

bdelete

```
[num] bd[!] [num]
```

Unload buffer *num* and remove it from the buffer list. Add a ! to force removal of an unsaved buffer. The buffer may also be specified by filename. If no buffer is specified, remove the current buffer. {Vim}

buffer

```
[num] b[!] [num]
```

Begin editing buffer *num* in the buffer list. Add a ! to force a switch from an unsaved buffer. The buffer may also be specified by filename. If no buffer is specified, continue editing the current buffer. {Vim}

buffers

```
buffers[!]
```

Print the members of the buffer list. Some buffers (e.g., deleted buffers) are not listed. Add ! to show unlisted buffers. ls is another abbreviation for this command. {Vim}

cd

```
cd dir
chdir dir
```

Change the editor's current directory to *dir*.

---

center

```
[address] ce [width]
```

Center the line within the specified *width*. If *width* is not specified, use textwidth.
{Vim}

---

change

```
[address] c[!]
text
.
```

Replace (change) the specified lines with *text*. Add a ! to switch the autoindent
setting during input of *text*. Terminate the input by entering a line consisting of just a
period.

---

close

```
clo[!]
```

Close the current window unless it is the last window. If the buffer in the window is
not open in another window, unload it from memory. This command will not close a
buffer with unsaved changes, but you may add ! to hide it instead. {Vim}

---

copy

```
[address] co destination
```

Copy the lines included in *address* to the specified *destination* address. The command
t (short for "to") is a synonym for copy.

Example

    :1,10 co 50   *Copy first 10 lines to just after line 50*

---

cquit

    cq[!]

Quit Vim with an error code. This is useful with Bash's "invoke an external editor
for the command line" feature, when you don't want Bash to execute the text in the
editing buffer. {Vim}

---

delete

    [*address*] d [*register*] [*count*]

Delete the lines included in *address*. If *register* is specified, save or append the text to
the named register. Register names are the lowercase letters a–z. Uppercase names
append text to the corresponding register. If *count* is specified, delete that many lines.

Examples

    :/Part I/,/Part II/-1d   *Delete to the line above "Part II"*
    :/main/+d   *Delete the line below "main"*
    :.,$d x   *Delete from this line to the last line into register* x

---

edit

    e[!] [+*num*] [*filename*]

Begin editing *filename*. If no *filename* is given, start over with a copy of the current
file. Add a ! to edit the new file even if the current file has not been saved since the
last change. With the +*num* argument, begin editing on line *num*. Alternatively, *num*
may be a pattern, of the form /*pattern*.

Examples

```
:e file            Edit file in the current editing buffer
:e +/^Index #      Edit the alternate file at the first line matching the given pattern
:e!                Start over again on the current file
```

file

```
f [filename]
```

Change the filename for the current buffer to *filename*. The next time the buffer is written, it will be written to file *filename*. When the name is changed, the buffer's "not edited" flag is set, to indicate that you are editing a nonexistent file. If the new filename is the same as a file that already exists on the disk, you will need to use `:w!` to overwrite the existing file. When specifying a filename, the `%` character can be used to indicate the current filename. A `#` can be used to indicate the alternate filename. If no *filename* is specified, print the current filename and status of the buffer.

Example

```
:f %.new
```

fold

```
address fo
```

Fold the lines specified by *address*. A fold collapses several lines on the screen into one line, which can later be unfolded. It doesn't affect the text of the file. {Vim}

foldclose

```
[address] foldc[!]
```

Close folds in the specified *address*, or at the present address if none is specified. Add a `!` to close more than one level of folds. {Vim}

foldopen

```
[address] foldo[!]
```

Open folds in the specified *address*, or at the present address if none is specified. Add a ! to open more than one level of folds. {Vim}

global

```
[address] g[!]/pattern/[commands]
```

Execute *commands* on all lines that contain *pattern* or, if *address* is specified, on all lines within that range. If *commands* are not specified, print all such lines. Add a ! to execute *commands* on all lines *not* containing *pattern*. See also v, later in this list.

Examples

```
:g/Unix/p            Print all lines containing "Unix"
:g/Name:/s/tom/Tom/  Change "tom" to "Tom" on all lines containing "Name:"
```

hide

```
hid
```

Close the current window unless it is the last window, but do not remove the buffer from memory. This command is safe to use on an unsaved buffer. {Vim}

insert

```
[address] i[!]
text
.
```

Insert *text* at the line before the specified *address*, or at the present address if none is specified. Add a ! to switch the `autoindent` setting during input of *text*. Terminate the input of new text by entering a line consisting of just a period.

join

[*address*] j[!] [*count*]

Place the text in the specified range on one line, with whitespace adjusted to provide two space characters after a period (.), no space characters before a), and one space character otherwise. Add a ! to prevent whitespace adjustment.

Example

    :1,5j!   *Join the first five lines, preserving whitespace*

jumps

    ju

Print the jump list used with the CTRL-I and CTRL-O commands. The jump list is a record of most movement commands that skip over multiple lines. It records the position of the cursor before each jump. {Vim}

k

[*address*] k *char*

Same as mark; see mark, later in this list.

last

    la[!]

Edit the last file from the command-line argument list. {Vim}

left

[*address*] le [*count*]

Left-align the lines specified by *address*, or the current line if no address is specified. Indent the lines by *count* spaces. {Vim}

---

list

> [*address*] l [*count*]

Print the specified lines so that tabs display as `^I` and the ends of lines display as `$`. l is like a temporary version of `:set list`.

---

map

> map[!] [*string commands*]

Define a keyboard macro named *string* as the specified sequence of *commands*. *string* is usually a single character or the sequence #*num*, the latter representing a function key on the keyboard. Use a ! to create a macro for input mode. With no arguments, list the currently defined macros.

Examples

`:map K dwwP`	*Transpose two words*
`:map q :w^M:n^M`	*Write the current file; go to the next file*
`:map! + ^[bi(^[ea)`	*Enclose the previous word in parentheses*

Vim has `K` and `q` commands, which the example aliases would hide.

---

mark

> [*address*] ma *char*

Mark the specified line with *char*, a single lowercase letter. Same as k. Return later to the line with '*x* (apostrophe plus *x*, where *x* is the same as *char*). Vim also uses uppercase and numeric characters for marks. Lowercase letters work the same as in vi. Uppercase letters are associated with filenames and can be used between multiple

files. Numbered marks, however, are maintained in a special *.viminfo* file and cannot be set using this command.

marks

```
marks [chars]
```

Print a list of marks specified by *chars*, or all current marks if no *chars* are specified. {Vim}

Example

```
:marks abc          Print marks a, b, and c
```

mkexrc

```
mk[!] file
```

Create an *.exrc* file containing `set` commands for changed `ex` options and key mappings. This saves the current option settings, allowing you to restore them later. *file* defaults to *.exrc* in the current directory if not specified. {Vim}

move

```
[address] m destination
```

Move the lines specified by *address* to the *destination* address.

Example

```
:.,/Note/m /END/       Move a text block to after the line containing "END"
```

new

```
[count] new
```

Create a new window *count* lines high with an empty buffer. {Vim}

next

```
n[!] [[+num] filelist]
```

Edit the next file from the command-line argument list. Use `args` to list these files. If *filelist* is provided, replace the current argument list with *filelist* and begin editing on the first file. With the *+num* argument, begin editing on line *num*. Alternatively, *num* may be a pattern, of the form */pattern*.

Example

```
:n chap*                Start editing all "chapter" files
```

nohlsearch

```
noh
```

Temporarily stop highlighting all matches to a search when using the `hlsearch` option. Highlighting is resumed with the next search. {Vim}

number

```
[address] nu [count]
   or
[address] # [count]
```

Print each line specified by *address*, preceded by its buffer line number. Use `#` as an alternate abbreviation for `number`. *count* specifies the number of lines to show, starting with *address*.

only

```
on [!]
```

Make the current window be the only one on the screen. Windows open on modified buffers are not removed from the screen (hidden), unless you also use the `!` character. {Vim}

open

> [*address*] o [/*pattern*/]

Enter open mode (vi) at the lines specified by *address*, or at the lines matching *pattern*. Exit open mode with Q. Open mode lets you use the regular vi commands, but only one line at a time. It may be useful on very distant internet ssh connections.

---

packadd

> pa[!] *packagename...*

Search for a plugin directory matching *packagename* and load the plugin. See the Vim help for details. {Vim}

---

preserve

> pre

Save the current editor buffer as though the system were about to crash.

---

previous

> prev[!]

Edit the previous file from the command-line argument list. {Vim}

---

print

> [*address*] p [*count*]

Print the lines specified by *address*. *count* specifies the number of lines to print, starting with *address*. P is another abbreviation.

Example

    `:100;+5p`   *Show line 100 and the next five lines*

---

put

    [*address*] `pu` [*char*]

Place previously deleted or yanked lines from the named register specified by *char* to the line specified by *address*. If *char* is not specified, the last deleted or yanked text is restored.

---

qall

    `qa[!]`

Close all windows and terminate the current editing session. Use ! to discard changes made since the last save. {Vim}

---

quit

    `q[!]`

Terminate the current editing session. Use ! to discard changes made since the last save. If the editing session includes additional files in the argument list that were never accessed, quit by typing `q!` or by typing `q` twice. Vim closes the editing window only if there are still other windows open on the screen.

---

read

    [*address*] `r` *filename*

Copy the text of *filename* after the line specified by *address*. If *filename* is not specified, the current filename is used.

Example

    `:0r $HOME/data`   *Read the named file in at the top of the current file*

---

read

```
[address] r !command
```

Read the output of *command* into the text after the line specified by *address*.

Example

```
:$r !spell %
```
                        Place the results of spellchecking at the end of the file

recover

```
rec [file]
```

Recover *file* from the system save area.

redo

```
red
```

Restore last undone change. Same as CTRL-R. {Vim}

resize

```
res [[±]num]
```

Resize the current window to be *num* lines high. If + or - is specified, increase or decrease the current window height by *num* lines. {Vim}

rewind

```
rew[!]
```

Rewind the argument list and begin editing the first file in the list. Add a ! to rewind the list even if the current file has not been saved since the last change.

right

```
[address] ri [width]
```

Right-align lines specified by *address*, or the current line if no address is specified, to column *width*. Use the value of the `textwidth` option if no *width* is specified. {Vim}

sbnext

```
[count] sbn [count]
```

Split the current window and begin editing the *count* next buffer from the buffer list. If no count is specified, edit the next buffer in the buffer list. {Vim}

sbuffer

```
[num] sb [num]
```

Split the current window and begin editing buffer *num* from the buffer list in the new window. The buffer to be edited may also be specified by filename. If no buffer is specified, open the current buffer in the new window. {Vim}

set

```
se parameter1 parameter2...
```

Set a value to an option with each *parameter*, or if no *parameter* is supplied, print all options that have been changed from their defaults. For Boolean options, each *parameter* can be phrased as *option* or no*option*; other options can be assigned with the syntax *option*=*value*. Specify `all` to list the current settings. The form `set` *option*? displays the value of *option*. See the tables that list `set` options in Appendix B.

Examples

```
:set nows wm=10
:set all
```

shell

```
sh
```

Create a new shell. Resume editing when the shell terminates.

---

snext

```
[count] sn [[+num] filelist]
```

Split the current window and begin editing the next file from the command-line argument list. If *count* is provided, edit the *count* next file. If *filelist* is provided, replace the current argument list with *filelist* and begin editing the first file. With the *+n* argument, begin editing on line *num*. Alternatively, *num* may be a pattern of the form **/***pattern*. {Vim}

---

source

```
so file
```

Read (source) and execute **ex** commands from *file*.

Example

```
:so $HOME/.exrc
```

---

split

```
[count] sp [+num] [filename]
```

Split the current window and load *filename* in the new window, or load the same buffer in both windows if no file is specified. Make the new window *count* lines high, or if *count* is not specified, split the window into equal parts. With the *+n* argument, begin editing on line *num*. *num* may also be a pattern of the form **/***pattern*. {Vim}

sprevious

    [*count*] spr [+*num*]

Split the current window and begin editing the previous file from the command-line argument list in the new window. If *count* is specified, edit the *count* previous file. With the +*num* argument, begin editing on line *num*. *num* may also be a pattern of the form /*pattern*. {Vim}

stop

    st

Suspend the editing session. Same as CTRL-Z. Use the shell `fg` command to resume the session.

substitute

    [*address*] s [/*pattern*/*replacement*/] [*options*] [*count*]

Replace the first instance of *pattern* on each of the specified lines with *replacement*. If *pattern* and *replacement* are omitted, repeat the last substitution. *count* specifies the number of lines on which to substitute, starting with *address*.

Options

c   Prompt for confirmation before each change.

g   Substitute all instances of *pattern* on each line (global).

p   Print the last line on which a substitution was made.

Examples

`:1,10s/yes/no/g`	*Substitute on the first 10 lines*
`:%s/[Hh]ello/Hi/gc`	*Confirm global substitutions*
`:s/Fortran/\U&/ 3`	*Uppercase "Fortran" on the next three lines*
`:g/^[0-9][0-9]*/s//Line &:/`	*For every line beginning with one or more digits, add "Line" and a colon*

suspend

> su

Suspend the editing session. Same as ⌈CTRL-Z⌋. Use the shell `fg` command to resume the session.

---

sview

> [*count*] sv [+*num*] [*filename*]

Same as the `split` command, but set the `readonly` option for the new buffer. {Vim}

---

t

> [*address*] t *destination*

Copy the lines included in *address* to the specified *destination* address. `t` is equivalent to `copy` and is short for "to."

Example

```
:%t$                    Copy the file and add it to the end
```

---

tag

> [*address*] ta *tag*

In the *tags* file, locate the file and line matching *tag* and start editing there.

Example

Run `ctags`, then switch to the file containing `main`:

```
:!ctags *.c
:tag main
```

---

tags

    tags

Print the list of tags in the tag stack. {Vim}

---

unabbreviate

    una *word*

Remove *word* from the list of abbreviations.

---

undo

    u

Reverse the changes made by the last editing command. In vi the undo command undoes itself, redoing what you undid. Vim supports multiple levels of undo. Use redo to redo an undone change in Vim.

---

unhide

    [*count*] unh

Split the screen to show one window for each active buffer in the buffer list. If specified, limit the number of windows to *count*. {Vim}

---

unmap

    unm[!] *string*

Remove *string* from the list of keyboard macros. Use ! to remove a macro for input mode.

---

v

> [*address*] v/*pattern*/[*command*]

Execute *command* on all the lines *not* containing *pattern*. If *command* is not specified, print all such lines. v is equivalent to g!. See global, earlier in this list.

Example

> :v/#include/d   *Delete all lines except "#include" lines*

---

version

> ve

Print the editor's version information.

---

view

> vie [+*num*] [*filename*]

Same as edit, but set the file to readonly. When executed in ex mode, return to normal or visual mode. {Vim}

---

visual

> [*address*] vi [*type*] [*count*]

Enter visual mode (vi) at the line specified by *address*. Return to ex mode with Q. *type* can be one of -, ^, or . (see the z command, later in this section). *count* specifies an initial window size.

---

visual

```
vi [+num] file
```

Begin editing *file* in visual mode (vi), optionally at line *num*. Alternatively, *num* may be a pattern of the form */pattern*. {Vim}

---

vsplit

```
[count] vs [+num] [filename]
```

Same as the split command, but split the screen vertically. The *count* argument can be used to specify a width for the new window. {Vim}

---

wall

```
wa[!]
```

Write all changed buffers with filenames. Add ! to force writing of any buffers marked readonly. {Vim}

---

wnext

```
[count] wn[!] [[+num] filename]
```

Write the current buffer and open the next file in the argument list, or the *count* next file if specified. If *filename* is specified, edit it next. With the +*num* argument, begin editing on line *num*. *num* may also be a pattern of the form */pattern*. Add ! to force writing of any buffers marked readonly. {Vim}

---

wq

```
wq[!]
```

Write and quit the file in one action. The file is always written. The ! flag forces the editor to write over any current contents of *file*.

wqall

    wqa[!]

Write all changed buffers and quit the editor. Add ! to force writing of any buffers marked `readonly`. `xall` is an alias for this command. {Vim}

write

    [address] w[!] [[>>] file]

Write the lines specified by *address* to *file*, or write the full contents of the buffer if *address* is not specified. If *file* is also omitted, save the contents of the buffer to the current filename. If >> *file* is used, append lines to the end of the specified *file*. Add a ! to force the editor to write over any current contents of *file*.

Examples

    :1,10w name_list   *Copy the first 10 lines to the file* name_list
    :50w >> name_list   *Now append line 50*

write

    [address] w !command

Write the lines specified by *address* to *command*.

Example

    :1,66w !pr -h myfile | lpr   *Print the first page of the file*

X

    X

Prompt for an encryption key. This can be preferable to `:set key`, as typing the key is not echoed to the console. To remove an encryption key, just reset the `key` option to an empty value. {Vim}

---

xit

    x

Write the file if it was changed since the last write, and then quit.

---

yank

    [*address*] y [*char*] [*count*]

Place the lines specified by *address* in the named register *char*. Register names are the lowercase letters `a–z`. Uppercase names append text to the corresponding register. If no *char* is given, place the lines into the general register. *count* specifies the number of lines to yank, starting with *address*.

Example

    :101,200 ya a   *Copy lines 101–200 to register* a

---

z

    [*address*] z [*type*] [*count*]

Print a window of text with the line specified by *address* at the top. *count* specifies the number of lines to be displayed.

Type

+

  Place the specified line at the top of the window (default).

-

  Place the specified line at the bottom of the window.

.

  Place the specified line in the center of the window.

^

Print the previous window.

=

Place the specified line in the center of the window and leave the current line at this line.

&

[*address*] & [*options*] [*count*]

Repeat the previous substitute (s) command. *count* specifies the number of lines on which to substitute, starting with *address*. *options* are the same as for the substitute command.

Examples

:s/Overdue/Paid/	*Substitute once on the current line*
:g/Status/&	*Redo the substitution on all "Status" lines*
:g/Status/&g	*Redo the substitution on all "Status" lines globally*

@

[*address*] @ [*char*]

Execute the contents of the register specified by *char*. If *address* is given, move the cursor to the specified address first. If *char* is @, repeat the last @ command.

=

[*address*] =

Print the line number of the line indicated by *address*. The default is the line number of the last line.

!

      `[address] !command`

Execute *command* in a shell. If *address* is specified, use the lines specified by *address* as standard input to *command*, and replace those lines with the output and error output. This is called *filtering* the text through the *command*.

Examples

`:!ls`	*List files in the current directory*
`:11,20!sort -f`	*Sort lines 11–20 of the current file*

< >

      `[address] < [count]`
        `or`
      `[address] > [count]`

Shift lines specified by *address* either left (<) or right (>). Only leading spaces and tabs are added or removed when shifting lines. *count* specifies the number of lines to shift, starting with *address*. The `shiftwidth` option controls the number of columns that are shifted. Repeating the < or > increases the shift amount. For example, `:>>>` shifts three times as much as `:>`.

~

      `[address] ~ [count]`

Replace the last-used regular expression (even if from a search and not from an `s` command) with the replacement pattern from the most recent `s` (substitute) command. This is rather obscure; see Chapter 6 for details.

---

address

      `address`

Print the lines specified by *address*.

ENTER

Print the next line in the file. (For ex only, not from the : prompt in vi.)

Setting Options

This appendix describes the important `set` command options for the "Heirloom" `vi`, Solaris `/usr/xpg7/bin/vi`, and Vim 8.2.

Heirloom and Solaris vi Options

Table B-1 contains brief descriptions of the important `set` command options. In the first column, options are listed in alphabetical order; if the option can be abbreviated, that abbreviation is shown in parentheses. The second column shows the default setting that `vi` uses unless you issue an explicit `set` command (either manually or in the *.exrc* file). The last column describes what the option does when enabled.

Table B-1. "Heirloom" and Solaris `vi` `set` options

Option	Default	Description
autoindent (ai)	noai	In insert mode, indent each line to the same level as the line above or below. Use with the `shiftwidth` option.
autoprint (ap)	ap	Change the display after each editor command. For global replacement, display the last replacement.
autowrite (aw)	noaw	Automatically write (save) the file if changed before opening another file with `:n` or before giving a Unix command with `:!`.
beautify (bf)	nobf	Ignore all control characters during input (except tab, newline, or form feed).
directory (dir)	*/var/tmp*	Names the directory in which ex/`vi` stores buffer files. The directory must be writable.
edcompatible	noedcompatible	Remember the flags used with the most recent substitute command (global, confirming), and use them for the next substitute command. Despite the name, no version of ed actually does this.
errorbells (eb)	noerrorbells	Sound the bell when an error occurs.

Option	Default	Description
exrc (ex)	noexrc	Allow the execution of *.exrc* files that reside outside the user's home directory.
flash (fp)	fp	Flash the screen instead of ringing the bell.
hardtabs (ht)	8	Define boundaries for terminal hardware tabs.
ignorecase (ic)	noic	Disregard case during a search.
lisp	nolisp	Insert indents in appropriate Lisp format. ( ), { }, [[, and]] are modified to have meaning for Lisp.
list	nolist	Print tabs as ^I; mark ends of lines with $.
magic	magic	Wildcard characters . (dot), * (asterisk), and [ ] (brackets) have special meaning in patterns.
mesg	mesg	Permit system messages to display on the terminal while editing in vi.
novice	nonovice	Require the use of long ex command names, such as copy or read. Solaris vi only.
number (nu)	nonu	Display line numbers on the left side of the screen during an editing session.
open	open	Allow entry to *open* or *visual* mode from ex. Although not in Solaris vi, this option has traditionally been in vi, and it may be in your Unix's version of vi.
optimize (opt)	noopt	Abolish carriage returns at the end of lines when printing multiple lines; this speeds output on dumb terminals when printing lines with leading whitespace (spaces or tabs).
paragraphs (para)	IPLPPPQP LIpplpipbp	Define paragraph delimiters for movement by { or }. The pairs of characters in the value are the names of troff macros that begin paragraphs.
prompt	prompt	Display the ex prompt (:) when vi's Q command is given.
readonly (ro)	noro	Any writes (saves) of a file fail unless you use ! after the write (works with w, ZZ, or autowrite).
redraw (re)		Redraw the screen whenever edits are made (in other words, insert mode pushes over existing characters, and deleted lines immediately close up). The default depends on line speed and terminal type. noredraw is useful at slow speeds on a dumb terminal: deleted lines show up as @, and inserted text appears to overwrite existing text until you press ESC. This option is essentially obsolete; let vi choose how to set it.
remap	remap	Allow nested map sequences.
report	5	Display a message on the status line whenever you make an edit that affects at least a certain number of lines. For example, 6dd reports the message "6 lines deleted."
scroll	[½ *window*]	Number of lines to scroll with ^D and ^U commands.
sections (sect)	SHNHH HU	Define section delimiters for [[and]] movement. The pairs of characters in the value are the names of troff macros that begin sections.

Option	Default	Description
shell (sh)	*/bin/sh*	Pathname of the shell used for shell escapes (`:!`) and the shell command (`:sh`). The default value is derived from the shell environment, which varies on different systems but often ends up being */bin/sh*.
shiftwidth (sw)	8	Define the number of spaces in backward (^D) tabs when using the `autoindent` option, and for the `<<` and `>>` commands.
showmatch (sm)	nosm	In `vi`, when `)` or `}` is entered, move the cursor briefly to the matching `(` or `{`. (If no match, ring the error message bell.) Very useful for programming.
showmode	noshowmode	In insert mode, display a message on the prompt line indicating the type of insert you are making, for example, "OPEN MODE" or "APPEND MODE."
slowopen (slow)		Hold off display during insert. The default depends on line speed and terminal type.
sourceany	nosourceany	Allow reading *.exrc* files that are not owned by the current user. "Heirloom" `vi` only.
tabstop (ts)	8	Define the number of spaces that a tab indents during an editing session. (Printers still use the system tab of 8.)
taglength (tl)	0	Define the number of characters that are significant for tags. The default (zero) means that all characters are significant.
tags	*tags /usr/lib/tags*	Define the pathname of files containing tags. See the Unix `ctags` command. By default, `vi` searches the file *tags* in the current directory and */usr/lib/tags*.
tagstack	tagstack	Enable stacking of tag locations on a stack. Solaris `vi` only.
term		Set the terminal type.
terse	noterse	Display shorter error messages. Ironically, there is no abbreviation for this option.
timeout (to)	timeout	Keyboard maps time out after 1 second.[a]
ttytype		Set the terminal type. This is just another name for `term`.
warn	warn	Display the warning message "No write since last change."
window (w)		Show a certain number of lines of the file on the screen. The default depends on line speed and terminal type.
wrapmargin (wm)	0	Define the right margin. If greater than zero, automatically insert carriage returns to break lines.
wrapscan (ws)	ws	Wrap searches around either end of the file.
writeany (wa)	nowa	Allow saving to any file.

[a] When you have mappings of several keys (for example, `:map zzz 3dw`), you probably want to use `notimeout`. Otherwise, you need to type zzz within one second. When you have an insert mode mapping for a cursor key (for example, `:map! ^[OB ^[ja`), you should use `timeout`. Otherwise, `vi` won't react to ESC until you type another key.

Vim 8.2 Options

In the preceding section, we listed all 46 "Heirloom" and Solaris `set` command options. Vim 8.2 has more than 400 (!) `set` command options. Table B-2 lists the options we consider to be most useful.

Most options described in Table B-1 are not repeated here.

The summaries in this table are by necessity very brief. Much more information about each option may be found in Vim's online help file *options.txt*.

Table B-2. Vim 8.2 set options

Option	Default	Description
autoread (ar)	noautoread	Detect whether a file inside Vim has been modified externally, not by Vim, and automatically refresh the Vim buffer with the changed version of the file.
background (bg)	dark or light	Vim tries to use background and foreground colors that are appropriate to the particular terminal. The default depends on the current terminal or windowing system.
backspace (bs)	0	Control whether you can backspace over a newline and/or over the start of insert. Values are 0 for `vi` compatibility; 1 to backspace over newlines and indents; and 2 to backspace over the start of newlines, insert, and indents.
backup (bk)	nobackup	Make a backup before overwriting a file and then leave it around after the file has been successfully written. To have a backup file just while the file is being written, use the `writebackup` option. See also `writebackup`.
backupdir (bdir)	., ~/tmp/, ~/	A list of directories for the backup file, separated with commas. The backup file is created in the first directory in the list where this is possible. If empty, you cannot create a backup file. The name `.` (dot) means the same directory as where the edited file is.
backupext (bex)	~	The string that is appended to a filename to make the name of the backup file.
binary (bin)	nobinary	Change a number of other options to make it easier to edit binary files. The previous values of these options are remembered and restored when `bin` is switched back off. Each buffer has its own set of saved option values. This option should be set before editing a binary file. You can also use the `-b` command-line option.

Option	Default	Description
breakat (brk)	" ^I!@*-+;:,./?"	Break line at any character in breakat string if the set option linebreak is on. See also options breakindent, linebreak, and showbreak for customizing this feature.
breakindent (bri)	nobreakindent	Indent lines wrapped by the breakat option.
cdpath (cd)	Same as value in environment variable CDPATH	A list of directories Vim will search with ex command cd or lcd in the same way $CDPATH works in the shell. If you use it in your shell, this will be familiar behavior.
cindent (cin)	nocindent	Enable automatic smart C program indenting.
cinkeys (cink)	0{,0},:,0#,!^F, o,O,e	A list of keys that, when typed in insert mode, cause reindenting of the current line. Only happens if cindent is on.
cinoptions (cino)		Affects the way cindent reindents lines in a C program. See the online help for details.
cinwords (cinw)	if, else, while, do, for, switch	These keywords start an extra indent in the next line when smartindent or cindent is set. For cindent, this is done only at an appropriate place (inside {…}).
cmdwinheight (cwh)	number (default 7)	Number of lines in the command-line window.
colorcolumn (cc)	empty string	Highlight columns listed in comma-separated list. This is a useful visual for vertical text alignment.
columns (co)	80 or terminal width	Usually set by Vim. Can be useful to define for your GUI instance at launch if you have a preference (as one of us does). See also lines.
comments (com)	s1:/*,mb:*,ex:*/,://, b:#,:%,:XCOMM,n:>,fb:-	A comma-separated list of strings that can start a comment line. See the online help for details.
compatible (cp)	cp; nocp when a *.vimrc* or Vim runtime *defaults.vim* file is found	Makes Vim behave more like vi in too many ways to describe here. It is on by default, to avoid surprises. Having a *.vimrc* turns off the vi compatibility; usually this is a desirable side effect.[a]
completeopt (cot)	menu,preview	A comma-separated list of options for insert mode completion.
cpoptions (cpo)	aABceFs	A sequence of single character flags, each one indicating a different way in which Vim will or will not exactly mimic vi. When empty, the Vim defaults are used. See the online help for details.
cursorcolumn (cuc)	nocursorcolumn	Highlight the screen column of the cursor with CursorColumn highlighting. This is useful for lining up text vertically. Can slow down the screen display.

Option	Default	Description
cursorline (cul)	nocursorline	Highlight the screen line of the cursor with Cursor Row highlighting. Makes it easy to find the current line in the edit session. Use in conjunction with cursorcolumn for a crosshairs effect. Can slow down the screen display.
cursorlineopt (culopt)	string, ""	Define the behavior of cursorline (cursor line must be set for this to have any effect). The most useful effect is to set it to number. This highlights line numbers only. While it can be useful to highlight the entire line, doing so becomes confusing when used together with syntax coloring, as the highlight alters the color and background of lines.
define (def)	^#\s*define	A search pattern that describes macro definitions. The default value is for C programs. For C++, use ^\(#\s*define\ |[a-z]*\s*const\s*[a-z]*\). When using the :set command, you need to double the backslashes.
dictionary (dict)	empty string	Comma-separated list of filenames to use for key word completion.
digraph (dg)	nodigraph	Useful for entering digraphs with *character1*, BACKSPACE , *character2*. See the discussion in "Digraphs: Non-ASCII Characters" on page 320.
directory (dir)	., ~/tmp, /tmp	A list of directory names for the swap file, separated with commas. The swap file will be created in the first directory where this is possible. If empty, no swap file will be used and recovery is impossible! The name . (dot) means to put the swap file in the same directory as the edited file. Using . first in the list is recommended so that editing the same file twice will result in a warning.
equalprg (ep)		External program to use for = command. When this option is empty, the internal formatting functions are used.
errorfile (ef)	errors.err	Name of the error file for the quickfix mode. When the -q command-line argument is used, error file is set to the following argument.
errorformat (efm)	(Too long to print)	scanf-like description of the format for the lines in the error file.
expandtab (et)	noexpandtab	When inserting a tab, expand it to the appropriate number of spaces.
fileformat (ff)	unix	Describes the convention to terminate lines when reading/writing the current buffer. Possible values are dos (CR/LF), unix (LF), and mac (CR). Vim usually sets this automatically.

Option	Default	Description
fileformats (ffs)	dos,unix	List the line-terminating conventions that Vim tries to apply to a file when reading. Multiple names enable automatic end-of-line detection when reading a file.
fixendofline (fix eol)	boolean, on	This ensures that a proper newline is appended to the last line of a file upon writing it out. If you don't want this, be sure to turn it off. For example, you don't want this if you are editing a binary file.
formatoptions (fo)	Vim default: tcq; vi default: vt	A sequence of letters that describes how automatic formatting is to be done. See the online help for details.
gdefault (gd)	nogdefault	Cause the substitute command to change all instances.
guifont (gfn)		A comma-separated list of fonts to try when starting the GUI version of Vim.
hidden (hid)	nohidden	Hide the current buffer when it is unloaded from a window, instead of abandoning it.
history (hi)	Vim default: 20; vi default: 0	Control how many ex commands, search strings, and expressions are remembered in the command history. Set this to a high number. Computer memory is cheap! See the section "Moving into the Fast Lane" on page 343 for an example of leveraging command-line history.
hlsearch (hls)	nohlsearch	Highlight all matches of the most recent search pattern.
icon	noicon	Vim attempts to change the name of the icon associated with the window where it is running. Overridden by the iconstring option.
iconstring		String value used for the icon name of the window.
ignorecase (ic)	noignorecase	Ignore case in searches. See also smartcase.
include (inc)	^#\s*include	Define a search pattern for finding include commands. The default value is for C programs.
incsearch (is)	noincsearch	Enable incremental searching.
isfname (isf)	@,48-57,/,.,-,_, +,,,$,:,~	A list of characters that can be included in filenames and pathnames. Non-Unix systems have different default values. The @ character stands for any alphabetic character. It is also used in the other is *XXX* options, described next.
isident (isi)	@,48-57,_,192-255	A list of characters that can be included in identifiers. Non-Unix systems may have different default values.
iskeyword (isk)	@,48-57,_,192-255	A list of characters that can be included in keywords. Non-Unix systems may have different default values. Keywords are used in searching and recognizing with many commands, such as w, [i, and many more.
isprint (isp)	@,161-255	A list of characters that can be displayed.

Option	Default	Description
laststatus (ls)	2	Controls when the last window will have a status line. Zero is never, one is only if there are at least two windows, and two is always.
linebreak (lbr)	nolinebreak	Break a long line at the characters defined in breakat. Vim wraps the line to keep the whole line visible.
lines	24 or terminal height	Usually set by Vim. Can be useful if you use the GUI and prefer to define the number of lines when starting Vim. See also columns.
listchars (lcs)	eol:$	Customize what Vim displays when the list option is set. Useful for defining spaces as dots. (Even more refined is defining leading and trailing spaces with the lead:. and trail:. definitions: :set listchars+=lead:.,trail:..)
makeef (mef)	/tmp/vim##.err	The error filename for the :make command. Non-Unix systems have different default values. The ## is replaced by a number to make the name unique.
makeprg (mp)	make	The program to use for the :make command. % and # in the value are expanded.
matchpairs (mps)	(:),{:},[:]	Comma-separated definitions of colon-separated matching character pairs. These must be two different characters. It is useful to add <:> for HTML matching. :set matchpairs+="<:>"
modifiable (ma)	modifiable	When turned off, do not allow any changes in the buffer.
mouse	a for GUI, MS-DOS, and Win32	Enable the mouse in non-GUI versions of Vim. This works for MS-DOS, Win32, QNX pterm, and xterm. See the online help for details.
mousehide (mh)	nomousehide	Hide the mouse pointer during typing. Restores the pointer when the mouse is moved.
numberwidth (nuw)	vi default: 8; Vim default: 4	Define the number column width (set with number or relativenumber). Vim always uses the last position for a separating space. We recommend setting this to at least six.
paste	nopaste	Change a large number of options so that pasting into a Vim window with a mouse does not mangle the pasted text. Turning it off restores those options to their previous values. See the online help for details.
relativenumber (rnu)	norelativenumber	Number lines on the left side of the window relative to the current line. For example, the current line displays the correct line number, and all lines above and below show the offset from the current line. This can be useful for block commands, removing the need to count lines.
ruler (ru)	noruler	Show the line and column number of the cursor position.

Option	Default	Description
scrollbind (scb)	noscrollbind	Bind the current window to scroll with other windows that also have scrollbind set. Useful for diff comparisons.
scrolloff (so)	0 (5 in *defaults.vim*)	Set the minimum number of lines above or below the cursor when scrolling. Useful for forcing context lines around the current position. We like setting scrolloff to three.
scrollopt (sbo)	ver,jump	Define scrollbind behavior. ver binds vertical scrolling between scrollbind windows. See the Vim help for detailed discussion.
secure	nosecure	Disable certain kinds of commands in the startup file. Automatically enabled if you don't own the *.vimrc* and *.exrc* files.
shellpipe (sp)		The shell string to use for capturing the output from :make into a file. The default value depends on the shell.
shellredir (srr)		The shell string for capturing the output of a filter into a temporary file. The default value depends on the shell.
showbreak (sbr)	empty string	Insert this string in front of wrapped lines.
showcmd (sc)	Vim showcmd, Unix noshowcmd, defined also in *defaults.vim*	Show vi command mode commands as they are entered. Vim displays the command on the righthand side of the ex command mode line. For example, the vi command 5cw to change five words is progressively displayed as it is input. Useful for keeping track of a command as you build it.
showmode (smd)	Vim default: smd; vi default: nosmd	Put a message in the status line for insert, replace, and visual modes.
sidescroll (ss)	0	How many columns to scroll horizontally. The value zero puts the cursor in the middle of the screen.
smartcase (scs)	nosmartcase	Override the ignorecase option if the search pattern contains uppercase characters.
spell	nospell	Turn on spellchecking.
spelllang (spl)	en	A comma-separated list of spellchecking language files.
suffixes	*.bak,~,.o,.h,.info,.swp	When multiple files match a pattern during filename completion, the value of this variable sets a priority among them in order to pick the one Vim will use.
taglength (tl)	0	Define the number of characters that are significant for tags. Default (zero) means that all characters are significant.
tagrelative (tr)	Vim default: tr; vi default: notr	Filenames in a *tags* file from another directory are taken to be relative to the directory where the *tags* file is.

Option	Default	Description
tags (tag)	./tags,tags	Filenames for the :tag command, separated by spaces or commas. The leading ./ is replaced with the full path to the current file.
tildeop (top)	notildeop	Make the ~ command behave like an operator.
undolevels (ul)	1000	The maximum number of changes that can be undone. A value of 0 means vi compatibility: one level of undo and u undoes itself. Non-Unix systems may have different default values.
viminfo (vi)		Read the *viminfo* file upon startup, and write it upon exiting. The value is complex; it controls the different kinds of information that Vim will store in the file. See the online help for details.
writebackup (wb)	writebackup	Make a backup before overwriting a file. The backup is removed after the file is successfully written, unless the backup option is also on.

[a] Since Vim 8.0, Vim sets compatible off if a Vim runtime file *defaults.vim* or system-wide *defaults.vim* exists. This is a much better default behavior and addresses long-standing complaints and confusion when neophytes try Vim but see no Vim-specific behaviors.

The Lighter Side of vi

Sure, `vi` is user friendly. It's just particular about who it makes friends with.
  —Anonymous

This appendix touches on a broad range of `vi`-related subjects. It covers:

- Accessing the files described here and earlier in the book.
- The online `vi` tutorial referenced in Part I.
- The *vi Powered* logo for your website (and other logos).
- `vi`-related swag.
- The Vim clutch.
- Some of the unusual and amazing things people have done with `vi` over the years.
- The *Vi Lovers Home Page*.
- A *different* `vi` clone.
- A brief mention of `vi` versus Emacs.
- Some nice `vi`-related quotations.

Accessing the Files

Many of the bits and pieces we present here were once freely available on the internet. Alas, that is no longer true. To remedy this, we have created a GitHub repository with the various files. Simply clone *https://www.github.com/learning-vi/vi-files* to make your own copy of the repository:

```
git clone https://www.github.com/learning-vi/vi-files
```

Example Files

Some of the files used as examples in Part I, "vi and Vim Fundamentals", are in the *book_examples* directory.

Source for clewn

The `clewn` program, mentioned in the section "The Clewn GDB Driver" on page 384, is provided in the *clewn-1.15* directory. To build and install it, use the following straightforward recipe:

```
cd clewn-1.15
./configure
make
sudo make install
```

Online vi Tutorial

First up is Walter Zintz's online tutorial from *UnixWorld* magazine, which was mentioned several times in Part I.

This tutorial is long gone from its original site, but we managed to find a copy on the internet at *https://www.ele.uri.edu/faculty/vetter/Other-stuff/vi/009-index.html*. So that you won't have to rely on this site still being active after this book's publication, we have placed a copy into our GitHub repository.[1]

In our GitHub repo, the tutorial is in the directory *unix-world-tutorial*. If you use Firefox as your browser (for example), it should be enough to do:

```
$ cd unix-world-tutorial
$ firefox ./009-index.html &          Use the browser of your choice
```

The tutorial covers the following topics:

- Editor fundamentals
- Line-mode addresses
- The `g` (global) command
- The substitute command
- The editing environment (the `set` command, tags, and `EXINIT` and *.exrc*)
- Addresses and columns
- The replacement commands, `r` and `R`

---

[1] The web page footers in this copy point back to where we found them. That site may or may not be online when you look at our copy.

- Automatic indentation
- Macros

The tutorial includes quiz questions at the end of several sections that you can use to see how well you've absorbed the material in the tutorial. Or you can just try the questions directly to see how well we've done with this book!

vi Powered!

Next up is the *vi Powered* logo (Figure C-1). This is a small GIF file you can add to your personal web page to show that you used `vi` to create it.

Figure C-1. vi Powered!

The logo is in the directory *vi-powered* in the GitHub repository.

The original home page for the *vi Powered* logo (created by Antonio Valle) was *http://www.abast.es/~avelle/vi.html*. That page was written in Spanish and is long gone. There is now an English-language home page and it has instructions for adding the logo, which consist of several simple steps:

1. Download the logo. Get it from our GitHub repository, or enter *https://darryl.com/vipower.gif* into your (graphical) web browser, and then save the logo to a file, or use a command-line web retrieval utility, such as `wget`.

2. Add the following code to your web page in an appropriate place:

   ```
   <A HREF="https://darryl.com/vi.shtml">
   <IMG SRC="vipower.gif">
   </A>
   ```

   This puts the logo into your page and makes it into a hypertext link that, when selected, will go to the *vi Powered* home page. You may wish to add an `ALT="This Web Page is vi Powered"` attribute to the `<IMG>` tag, for users of nongraphical browsers.

3. Add the following code to the `<HEAD>` section of your web page:

   ```
   <META name="editor" content="/usr/bin/vi">
   ```

Just as the Real Programmer will eschew a WYSIWYG word processor in favor of troff, so too will Real Webmasters eschew fancy HTML authoring tools in favor of vi. You can use the *vi Powered* logo to display this fact with pride. ☺

You can find the Vim logo, in several variations, at *https://www.vim.org/logos.php*. A number of *Vim Powered* logos for websites are at *https://www.vim.org/buttons.php*.

vi for Java Lovers

Despite the title, this subsection is about the java you drink, not the Java you program in.

Our hypothetical Real Programmer, while using vi to write her C++ code, her troff documentation, and her web page, undoubtedly will want a cup of coffee now and then. She can now drink her coffee from a mug with a vi command reference printed on it!

So here's our third item: vi reference mugs, T-shirts, sweatshirts, barbecue aprons, baby bibs, and even mouse pads are all available from *https://www.cafepress.com/geek cheat/366808*.

The Vim Clutch

If changing modes in vi or Vim with your hands is bothersome, you may want to create your very own "Vim clutch." This USB-connected foot pedal sends i when pressed down and ESC when released.

The project is described, with parts, links, instructions, and photographs, at *https:// github.com/alevchuk/vim-clutch*.

Another Vim clutch is presented at *https://l-o-o-s-e-d.net/vim-clutch*. Along the way, the author describes several more Vim clutch projects besides his own.

Amaze Your Friends!

A collection of useful items related to vi were once upon a time available in the FTP archives at alf.uib.no. The original archives were at *ftp://afl.uib.no/pub/vi*. The collection was mirrored at *ftp://ftp.uu.net/pub/text-processing/vi*. Both of these sites are no longer available.

Happily, Clement Cole made his copy of the archive available to us for inclusion in our GitHub repository, for which we thank him.[2]

Unfortunately, these files were last updated in May 1995. Fortunately, vi's basic functionality has not changed, and much of the information and macros in the archive are still useful. The original archive had four subdirectories:

docs
> Documentation on vi, and also some comp.editors postings.

macros
> vi macros.

comp.editors
> Various materials posted to comp.editors.

programs
> Source code for vi clones for various platforms (and other programs).

We have not included the *programs* directory since it's largely irrelevant today.

The *docs* and *macros* are the most interesting. The *docs* directory has a large number of articles and references, including beginners' guides, explanations of bugs, quick references, and many short "how to" kinds of articles (e.g., how to capitalize just the first letter of a sentence in vi). There's even a song about vi!

The *macros* directory has more than 50 files in it that do different things. We mention just three of them here. Files whose original archived names ended in .tar.Z have been expanded into separate directories in our GitHub repository.

evi-tar
> An Emacs "emulator." The idea behind it is to turn vi into a modeless editor (one that is always in input mode, with commands done with control keys). It is actually done with a shell script that replaces the EXINIT environment variable.

hanoi
> This is perhaps the most famous of the unusual uses of vi: a set of macros that solve the Towers of Hanoi programming problem. This program simply displays the moves; it does not actually draw the disks. For fun, we have reprinted it in "The Towers of Hanoi, vi Version" on page 476.

---

2 Thanks also to Bakul Shah, who pointed us to the copy available online at *https://web.archive.org/web/19970209203017/http://archive.uwp.edu/pub/vi/*.

turing-tar

    This program uses vi to implement an actual Turing machine! It's rather amazing to watch it execute the programs.

There are many, many interesting macros in addition to these; take a look!

The Towers of Hanoi, vi Version

```
" From: gregm@otc.otca.oz.au (Greg McFarlane)
" Newsgroups: comp.sources.d,alt.sources,comp.editors
" Subject: VI SOLVES HANOI
" Date: 19 Feb 91 01:32:14 GMT
"
" Submitted-by: gregm@otc.otca.oz.au
" Archive-name: hanoi.vi.macros/part01
"
" Everyone seems to be writing stupid Tower of Hanoi programs.
" Well, here is the stupidest of them all: the hanoi solving
" vi macros.
"
" Save this article, unshar it, and run uudecode on
" hanoi.vi.macros.uu. This will give you the macro file
" hanoi.vi.macros.
" Then run vi (with no file: just type "vi") and type:
"       :so hanoi.vi.macros
"       g
" and watch it go.
"
" The default height of the tower is 7 but can be easily changed
" by editing the macro file.
"
" The disks aren't actually shown in this version, only numbers
" representing each disk, but I believe it is possible to write
" some macros to show the disks moving about as well. Any takers?
"
" (For maze solving macros, see alt.sources or comp.editors)
"
" Greg
"
" ----------- REAL FILE STARTS HERE --------------
set remap
set noterse
set wrapscan
" to set the height of the tower, change the digit in the following
" two lines to the height you want (select from 1 to 9)
map t 7
map! t 7
map L 1G/t^MX/^0^M$P1GJ$An$BGC0e$X0E0F$X/T^M@f^M@h^M$A1GJ@f0l$Xn$PU
map g IL
map I KMYNOQNOSkRTV
map J /^0[^t]*$^M
map X x
map P p
map U L
map A "fyl
map B "hyl
map C "fp
```

```
map e "fy2l
map E "hp
map F "hy2l
map K 1Go^[
map M dG
map N yy
map O p
map q tllD
map Y o0123456789Z^[0q
map Q 0iT^[
map R $rn
map S $r$
map T ko0^M0^M^M^[
map V Go/^[
```

The Vi Lovers Home Page

The Vi Lovers Home Page contains the following items:

- A table of all known `vi` clones, with links to the source code or binary distributions
- Links to other `vi` sites
- A large number of links to `vi` documentation, manuals, help, and tutorials, at a number of different levels
- `vi` macros for writing HTML documents and solving the Towers of Hanoi, and FTP sites for other macro sets
- Miscellaneous `vi` links: poems, a story about the "real history" of `vi`, `vi` versus Emacs discussions, and `vi` coffee mugs (see the section "vi for Java Lovers" on page 474)

Be aware that this site seems to have not been updated in a long time. Many of the links work, but many do not.

A Different vi Clone

Depicted in Figures C-2 through C-9 is the story of `vigor`, a *different* `vi` clone.
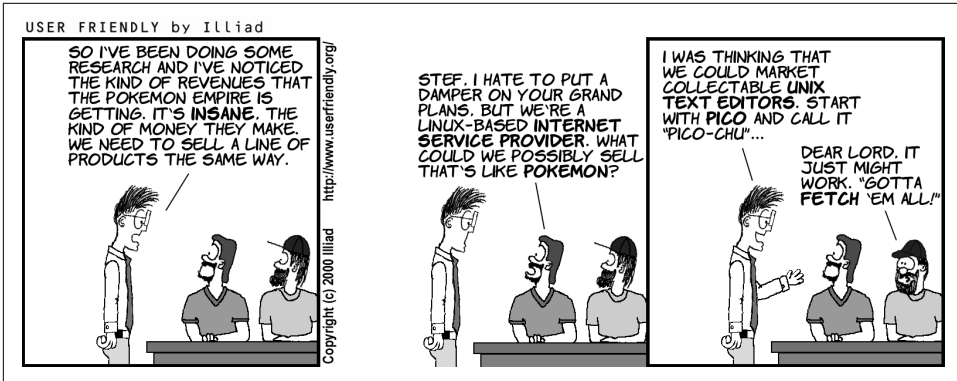
Figure C-2. The story of `vigor`*—part I*
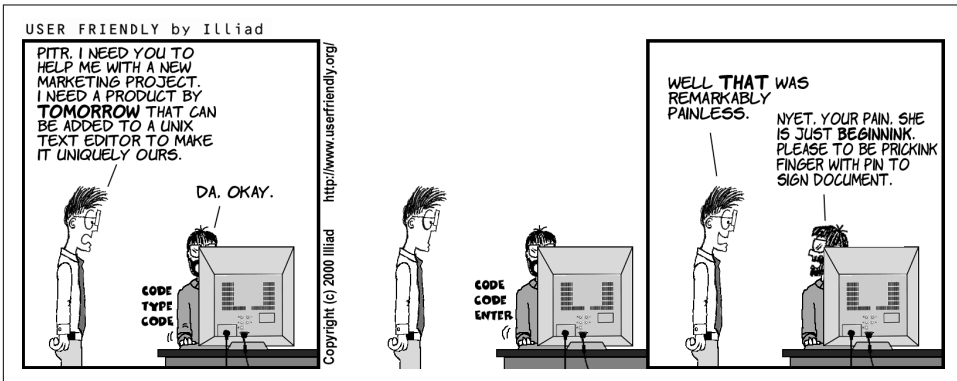
Figure C-3. The story of `vigor`*—part II*
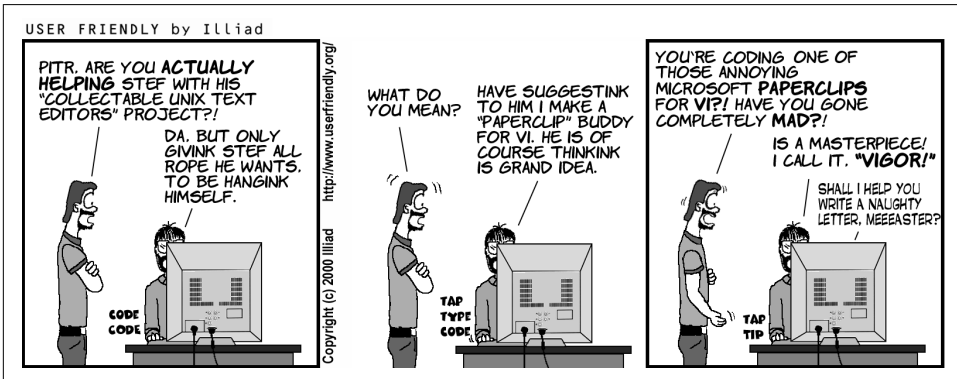
Figure C-4. The story of `vigor`*—part III*
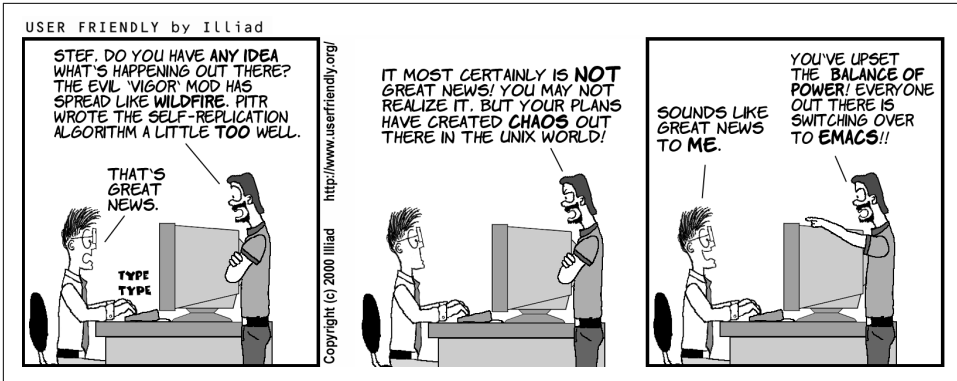
Figure C-5. The story of `vigor`—*part IV*
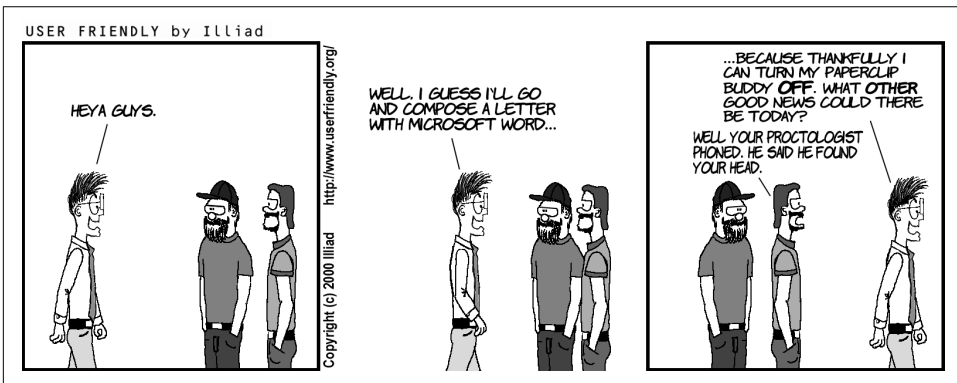
Figure C-6. The story of `vigor`—*part V*

Figure C-7. The story of `vigor`—*part VI*
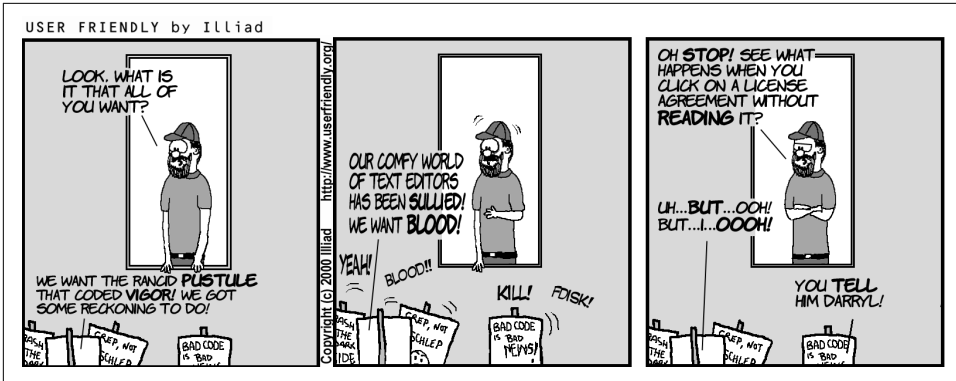
Figure C-8. The story of `vigor`*—part VII*
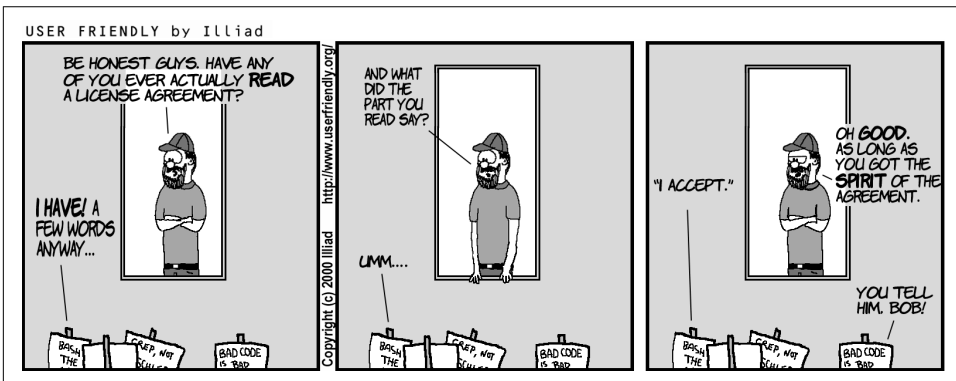
Figure C-9. The story of `vigor`*—part VIII*

The source code for `vigor` is available at *http://vigor.sourceforge.net*.

Tastes Great, Less Filling

```
vi is [[13~^[[15~^[[15~^[[19~^[[18~^ a
muk[^[[29~^[[34~^[[26~^[[32~^ch better editor than this emacs. I know
I^[[14~'ll get flamed for this but the truth has to be
said. ^[[D^[[D^[[D^[[D ^[[D^[^[[D^[[D^[[B^
exit ^X^C quit :x :wq dang it :w:w:w :x ^C^C^Z^D
```

— Jesper Lauridsen from `alt.religion.emacs`

We can't discuss `vi` as part of Unix culture without acknowledging what is perhaps the longest running debate in the Unix community: `vi` versus Emacs.[3]

---

3 OK, it's really a religious war, but we're trying to be nice. The other religious war, BSD versus System V, was settled by POSIX. System V won, although BSD received significant concessions. ☺

Discussions about which is better have cropped up on `comp.editors` (and other newsgroups) for years and years. (This is illustrated nicely in Figure C-10.)
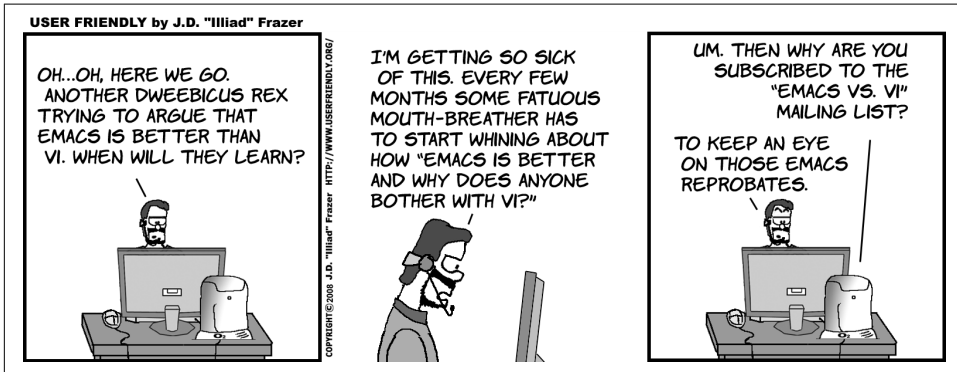
Figure C-10. It's not a religious war. Really!

Some of the better arguments in favor of `vi` are:

- `vi` is available on every Unix system. If you are installing systems or moving from system to system, you might have to use `vi` anyway.

- You can usually keep your fingers on the home row of the keyboard. This is a big plus for touch typists.

- Commands are one (or sometimes two) regular characters; they are much easier to type than all of the control and metacharacters that Emacs requires.

- `vi` is generally smaller and less resource intensive than Emacs. Startup times are appreciably faster, sometimes up to a factor of 10.

- Now that Vim (and other `vi` clones) has added features such as incremental searching, multiple windows and buffers, GUI interfaces, syntax highlighting and smart indenting, and programmability via extension languages, the functional gap between the two editors has narrowed significantly, if not disappeared entirely.

To be complete, one more item should be mentioned. Although GNU Emacs has always had `vi`-emulation packages, they were usually not very good. However, the `viper-mode` is reputed to be an excellent `vi` emulation. It can serve as a bridge for learning Emacs for those who are interested in doing so.

To conclude, always remember that you are the final judge of a program's utility. You should use the tools that make you the most productive, and for many tasks, `vi` and Vim are excellent tools.

vi Quotes

Finally, here are some more `vi` quotes, courtesy of Bram Moolenaar, Vim's author:

> THEOREM: `vi` is perfect.
>
> PROOF: VI in roman numerals is 6. The natural numbers less than 6 which divide 6 are 1, 2, and 3. 1 + 2 + 3 = 6. So 6 is a perfect number. Therefore, `vi` is perfect.
>
> —Arthur Tateishi

A reaction from Nathan T. Oelger:

> So, where does the above leave Vim? VIM in roman numerals might be: (1000 – (5 + 1)) = 994, which happens to be equal to 2*496+2. 496 is divisible by 1, 2, 4, 8, 16, 31, 62, 124, and 248 and 1+2+4+8+16+31+62+124+248 = 496. So, 496 is a perfect number. Therefore, Vim is twice as perfect as `vi`, *plus* a couple extra bits of goodies. ☺
>
> That is, Vim is *better* than perfect.

This quote seems to sum it up for the true `vi` lover:

> To me `vi` is zen. To use `vi` is to practice zen. Every command is a koan. Profound to the user, unintelligible to the uninitiated. You discover truth every time you use it.
>
> —Satish Reddy

vi and Vim: Source Code and Building

On the off chance that you don't already have `vi` or Vim installed on your system, this appendix describes where to get source code for both editors, and prebuilt installable binaries for most popular operating systems.

Nothing Like the Original

For many, many years, the source code to the original `vi` was unavailable without a Unix source code license. Although educational institutions were able to get licenses at relatively low cost, commercial licenses were always expensive. This fact prompted the creation of Vim and many other `vi` clones.

In January 2002, the source code for V7 and 32V UNIX was made available under an open source–style license.[1] This opened up access to almost all of the code developed for BSD Unix, including `ex` and `vi`.

The original code does not compile "out of the box" on modern systems, such as GNU/Linux, and porting it is difficult.[2] Fortunately, the work has already been done. If you would like to use the original, "real" `vi`, you can download the source code and build it yourself. See *https://github.com/n-t-roff/heirloom-ex-vi* for more information.

We were able to build the "Heirloom" `vi` with no problems on an Ubuntu GNU/Linux system just by following the instructions in the *README* file.

---

1 For more information about this, see the Unix Heritage Society website.

2 We know. We tried.

Where to Get Vim

Most modern Unix-flavored OSs use Vim as the standard version of vi.[3] That is, when you execute vi, you get Vim.

Many such systems lag slightly behind the most current Vim. For example, as of this publication (eighth edition, late 2021), the current release is Vim 8.2, and most systems have Vim 8.0.

In this section we briefly discuss how to install the latest version of Vim (or any version to your liking) on GNU/Linux (in this example, Ubuntu). For other GNU/Linux distributions, the process is basically the same.

If the command vi or vim doesn't start your editor, either it is not installed, or your path doesn't include Vim's executable directory. Make sure your PATH environment variable includes the following directories. (If this fails, it's possible Vim is not installed. Read on for instructions to install Vim.)

```
/usr/bin          This should be in your $PATH anyway
/bin              So should this
/opt/local/bin
/usr/local/bin
```

Verify your Vim version with the ex command version. Vim displays something like this:

```
VIM - Vi IMproved 8.2 (2019 Dec 12, compiled May  8 2021 05:44:12)
macOS version
Included patches: 1-2029
Compiled by root@apple.com
Normal version without GUI.  Features included (+) or not (-):
+acl                -farsi            +mouse_sgr          +tag_binary
-arabic             +file_in_path     -mouse_sysmouse     +tag_old_static
+autocmd            +find_in_path     -mouse_urxvt        -tag_any_white
+autochdir          +float            +mouse_xterm        -tcl
-autoservername     +folding          +multi_byte         -termguicolors
-balloon_eval       -footer           +multi_lang         +terminal
-balloon_eval_term  +fork()           -mzscheme           +terminfo
-browse             -gettext          +netbeans_intg      +termresponse
+builtin_terms      -hangul_input     +num64              +textobjects
+byte_offset        +iconv            +packages           +textprop
+channel            +insert_expand    +path_extra         +timers
+cindent            -ipv6             -perl               +title
-clientserver       +job              +persistent_undo    -toolbar
+clipboard          +jumplist         +popupwin           +user_commands
+cmdline_compl      -keymap           +postscript         -vartabs
+cmdline_hist       +lambda           +printer            +vertsplit
+cmdline_info       -langmap          -profile            +virtualedit
+comments           +libcall          +python/dyn         +visual
-conceal            +linebreak        -python3            +visualextra
```

---

3 The exceptions tend to be legacy Unix-based systems, such as HP/UX and AIX, where the standard vi is the original one.

```
+cryptv           +lispindent      +quickfix        +viminfo
+cscope           +listcmds        +reltime         +vreplace
+cursorbind       +localmap        -rightleft       +wildignore
+cursorshape      -lua             +ruby/dyn        +wildmenu
+dialog_con       +menu            +scrollbind      +windows
+diff             +mksession       +signs           +writebackup
+digraphs         +modify_fname    +smartindent     -X11
-dnd              +mouse           -sound           -xfontset
-ebcdic           -mouseshape      +spell           -xim
-emacs_tags       -mouse_dec       +startuptime     -xpm
+eval             -mouse_gpm       +statusline      -xsmp
+ex_extra         -mouse_jsbterm   -sun_workshop    -xterm_clipboard
+extra_search     -mouse_netterm   +syntax          -xterm_save
   system vimrc file: "$VIM/vimrc"
     user vimrc file: "$HOME/.vimrc"
 2nd user vimrc file: "~/.vim/vimrc"
     user exrc file: "$HOME/.exrc"
       defaults file: "$VIMRUNTIME/defaults.vim"
   fall-back for $VIM: "/usr/share/vim"
Compilation: gcc -c -I. -Iproto -DHAVE_CONFIG_H   -DMACOS_X_UNIX  -g -O2
-U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=1
Linking: gcc   -L/usr/local/lib -o vim  -lm -lncurses  -liconv -framework Cocoa
```

If you see the latest release/version, and that's what you wanted, you are done. If you want a different version, read on.

> Interestingly, on one of the authors' Mac mini, with OS X version 10.4.10 installed, not only does a vi command invoke Vim but the documentation (the "man page") references Vim!

If none of the preceding measures work, you probably don't have Vim. Vim is available in many forms for many platforms and is (usually) relatively easy to retrieve and install. The following sections guide you to getting Vim for your platform, in this order:

- Unix and variants, including GNU/Linux and Cygwin
- Windows XP and up
- Macintosh macOS

The installation procedure described here requires a development environment capable of compiling source code. Although most Unix variants provide compilers and related tools, some (notably current releases of the Ubuntu GNU/Linux distribution) require you to download and install additional packages before you can experience the pleasures of compiling code.

There are also prepackaged Vim bundles offering easy standard installations for GNU/Linux (Red Hat RPMs, Debian pkgs), Solaris (Companion Software), and HP-UX. The Vim home page provides links for all of these systems. For unusual systems, internet searching will undoubtedly prove helplful.

A quick check for `gcc` should indicate whether or not you are ready to compile Vim:

```
$ type gcc
gcc is /usr/bin/gcc
```

Getting Vim for Unix and GNU/Linux

Many modern Unix environments already come with some version of Vim. Most GNU/Linux distributions simply link the default `vi` location */usr/bin/vi* to a Vim executable. Most users won't ever need to install it. As mentioned earlier in the book, the Solaris 11 `vi` is actually Vim!

On Ubuntu GNU/Linux systems, a minimal version of Vim is installed as `vi`. For a full version, including GUI, do:

```
sudo apt install vim-gtk3
```

On other systems, you will need to do something similar with your system's package manager.

Because there are so many variants of Unix and so many flavors of some variants (e.g., Solaris, HP-UX, *BSD, all the distributions of GNU/Linux), if you can't install Vim by using a package manager, the next most straightforward way to get Vim is to download its source, compile it, and install it.

Vim is distributed as a compressed tar file (using either `gzip` or `bzip2` files—`.gz` and `.bz2`, respectively). Along with the tar file, each major version has a number of patches to fix problems or issues discovered after the release of each preceding major version.

It is possible to download the tar file and the patch files and then apply the patches individually in order to build from the latest source code. However, this process is tedious, as there are often hundreds of patch files for any given version.

Instead, it is much easier to simply *clone* the source code from Vim's Git repository on GitHub. Doing so gives output similar to this:

```
$ git clone git://github.com/vim/vim
Cloning into 'vim'...
remote: Enumerating objects: 34, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (27/27), done.
Receiving objects: 100% (113446/113446), 90.87 MiB | 1.07 MiB/s, done.
Resolving deltas: 100% (95729/95729), done.
Updating files: 100% (3347/3347), done.
```

To build, change to the *src* directory and run `configure`. You will probably want to do some internet research with respect to options to `configure` before running it. The output is voluminous:

```
$ cd vim/src
$ ./configure
configure: creating cache auto/config.cache
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
    ...
```

The next step is to run `make`. Here too, the output is voluminous:

```
$ make
/bin/sh install-sh -c -d objects
touch objects/.dirstamp
CC="gcc -Iproto -DHAVE_CONFIG_H        " srcdir=. sh ./osdef.sh
gcc -c -I. -Iproto -DHAVE_CONFIG_H     -g -O2 -U_FORTIFY_SOURCE
-D_FORTIFY_SOURCE=1       -o objects/arabic.o arabic.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H     -g -O2 -U_FORTIFY_SOURCE
-D_FORTIFY_SOURCE=1       -o objects/arglist.o arglist.c
gcc -c -I. -Iproto -DHAVE_CONFIG_H     -g -O2 -U_FORTIFY_SOURCE
-D_FORTIFY_SOURCE=1       -o objects/autocmd.o autocmd.c
    ...
```

When done, you'll have an executable name `vim`. To install it, become the `root` user and run `make install`. That's it!

Getting Vim for Windows Environments

MS Windows gvim

There are three main options for Microsoft Windows. The first is the self-installing executable *gvim82.exe*, available from the Vim home page. Download and run this, and it should do the rest. We have installed Vim using this executable on different Windows machines, and it's always worked cleanly. The binary should install correctly on all MS-Windows systems from Windows XP and later.

At one point in the install process, a DOS window pops up and gives a warning about something not being verifiable. We have never seen this become a problem.

Cygwin for Windows

The second option for Windows users is to install Cygwin, a suite of common GNU tools ported to the Windows platform. It's a full implementation of virtually all mainstream software used on Unix platforms. Vim is part of the standard Cygwin installation and runs in the Cygwin shell console.

Using Vim with Cygwin

The text-based console Vim works fine in Cygwin, but Cygwin's `gvim` expects an X Window System server to be running. It degrades gracefully into running the text-based Vim if started without this server.

To get Cygwin's `gvim` working (assuming you wish to run it on a local screen), start Cygwin's X server from the command line in a Cygwin shell as follows:

```
$ X -multiwindow &
```

The `-multiwindow` option tells the X server to let Windows manage the Cygwin applications. There are many other ways to use Cygwin's X server, but that discussion is outside the scope of this book. Installation of Cygwin's X server is also outside our scope here; if it is not installed, see the Cygwin home page for further information. A graphical "X" icon should appear in the Windows systray. This assures that the X server is in fact running.

It is confusing to have both Cygwin's Vim and *vim.org*'s Vim installed at the same time. Some of the configuration files referenced for Vim configuration may reside in different places, thus resulting in seemingly identical versions of Vim that start up with completely different options. For instance, Cygwin and Windows may have different notions of what is the home directory.

Windows Subsystem for Linux and Vim

The *Windows Subsystem for Linux* (WSL) is a virtual environment with full Linux kernel compatibility. It is Microsoft's platform for installing and running GNU/Linux distributions. The list of distributions is constantly growing, and most of the favorite GNU/Linux distributions are available.

For more discussion on WSL, see the section "Running gvim in Microsoft Windows WSL" on page 203.

WSL is a relatively recent addition to Microsoft Windows. While we've not described WSL in great detail, we consider it a better choice and recommend GNU/Linux in WSL and Vim over the previously mentioned Cygwin.

Getting Vim for the Macintosh Environment

There are two options for using Vim under macOS. You may use the native version, or you can install a graphical version using Homebrew. This section presents both options.

Native macOS Vim

macOS includes Vim as a standard tool. It's easiest to use Apple's default since OS updates also keep Vim up to date. Notably, macOS's Vim is not GUI, but there is a popular third-party GUI version of Vim, called MacVim, which we recommend.

MacVim is actively maintained and has a familiar Macintosh-like look and feel, as its maintainers adhere to common Macintosh styles and ergonomics.

At the bottom of MacVim's GitHub page, you'll find the *README.md* information shown in Figure D-1.
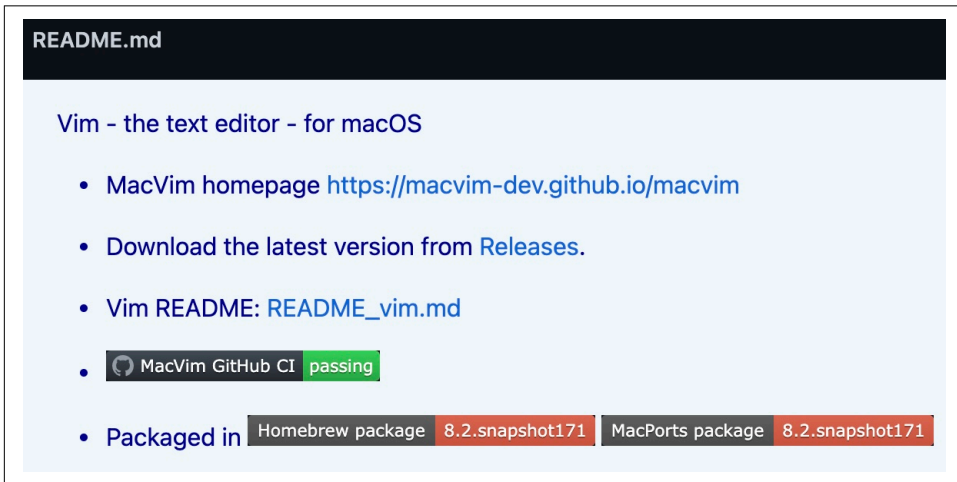
Figure D-1. MacVim home page, README.md

Click `Releases` from "Download the latest version from Releases". Near the bottom of that page, see `Assets`. (See Figure D-2.) We recommend downloading *MacVim.dmg* and performing a standard macOS install.
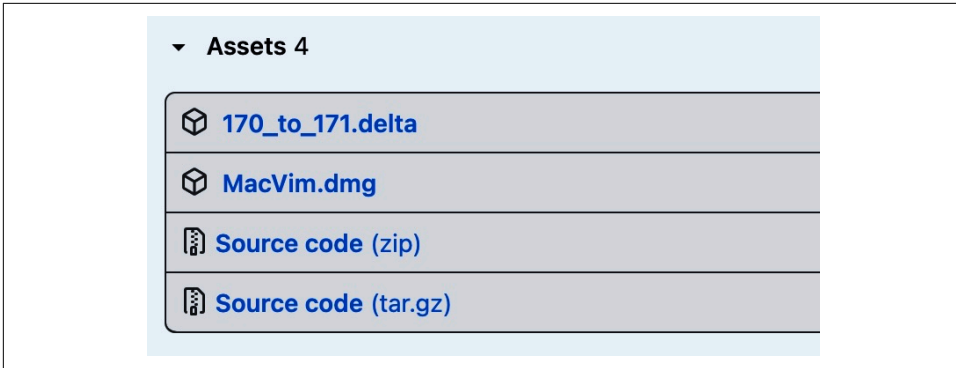
Figure D-2. MacVim assets (downloads)

One of us uses a `zsh` alias on his MacBook Pro by adding this line to his *.zshrc* profile:

```
alias vi='/Applications/MacVim.app/Contents/bin/mvim'
```

Installing Vim with Homebrew

Macintosh users who prefer a more GNU-like OS are probably familiar with Homebrew, an application manager that provides GNU packages for Macintosh computers.

Homebrew's GNU package install command is simply `brew install` *gnu-package*, where *gnu-package* is any available Homebrew package. To install Vim with Homebrew, execute the command:

```
brew install vim
```

Visit "Homebrew Formulae" for more detailed information about Homebrew Vim options.

Other Operating Systems

Vim's *vi_diff.txt* help file lists more environments for which Vim is supported. Read this file for more information. The environments include:

- IBM OS/390
- OpenVMS
- QNX

A number of now-obsolete systems were supported in the past, but they are likely no longer of interest.

Index

About the Authors

Arnold Robbins, an Atlanta native, is a professional programmer and technical author. He is also a happy husband, the father of four wonderful children, and an amateur Talmudist (Babylonian and Jerusalem). Since late 1997, he and his family have been living in Israel.

Arnold has been working with Unix systems since 1980, when he was introduced to a PDP-11 running a version of Sixth Edition Unix. His experience also includes multiple commercial Unix systems, from Sun, IBM, HP, and DEC. He has been working with GNU/Linux systems since 1996.

Arnold has also been a heavy `awk` user since 1987, when he became involved with `gawk`, the GNU project's version of `awk`. As a member of the POSIX 1003.2 balloting group, he helped shape the POSIX standard for `awk`. He is the longtime maintainer of `gawk` and its documentation.

In previous incarnations he has been a systems administrator and a teacher of Unix and networking continuing education classes. He has also had more than one poor experience with startup software companies, which he prefers not to think about anymore.

In Israel, he worked for several years writing high-end Command and Control–related software at a leading Israeli software company. He then spent a long time working as a software engineer at Intel and later at McAfee. These days he works at a smaller company that provides network security monitoring for the manufacturing and building management industries. His personal website may be found at *http://www.skeeve.com.*

O'Reilly has been keeping him busy: he is the author or coauthor of the bestselling titles *Unix in a Nutshell* (4th ed.), *Effective awk Programming* (4th ed.), *sed & awk* (2nd ed.) with Dale Dougherty, and *Classic Shell Scripting* with Nelson H. F. Beebe, and of several pocket references.

Elbert Hannah started out as a professional musician, changed direction, and chose computers and IT for his career. While `vi`, and later Vim, may not be the sole reason for the switch, one cannot discount the contribution "finding" `vi` made to his change in career choice.

Elbert finds particular joy knowing he writes and creates this book on Vim using Vim!

Starting as a music major in college, playing cello for a living, Elbert had a life-changing bicycling accident that changed his path forever. With no use of one hand, unable to play cello, Elbert temporarily chose mathematics as his major while

healing. Mathematics required computer science as a minor, and that was that. Elbert maintained his love for music and continues to play nonprofessionally today, but IT became his career.

Elbert discovered Unix while working in the telco industry, finding a remote job entry (RJE) linked to the IBM mainframe. He discovered that many processes were easier when they were diverted to the AT&T System V Unix computer by leveraging the myriad "specialized" (do one thing and do it well) commands for transformations and reports and then diverted back to the mainframe.

His early work with Unix required deep understanding of `ed`, and that set the stage for his long journey learning, loving, and evangelizing `vi` and eventually Vim. The seventh and eighth editions of *Learning the vi and Vim Editors* are Elbert's way of paying forward his appreciation and respect for Vim's impact in the coding universe.

Elbert specialized in integration of disparate systems. There are many users who use his applications, not aware that underneath are many separate applications. If you dig deep enough, you might find his picture on the cover of *CEO Magazine* (circa early to mid '90s) for his work integrating telco facilities and allocation applications.

Elbert developed an external web tool providing quick, easy, and enhanced ways to find information from a third party's product, which quickly established itself as a popular troubleshooting method for support teams and devops staff. He gave a presentation demonstrating the web tool in Las Vegas in 2018.

Elbert contributed to columns in more than one hundred technical publications (under a pseudonym) and was featured in a two-part column with Linus Torvalds, "The Great FOSS Debates: Kernel Truths" and "FOSS Debates, Part 2: Standard Deviations."

Colophon

The animal on the cover of *Learning the vi and Vim Editors*, eighth edition, is a tarsier, a nocturnal mammal related to the lemur. Its generic name, *Tarsius*, is derived from the animal's very long anklebone, the tarsus. Though they were once more widespread, the 10 species and 4 subspecies of the tarsier are now limited to the islands of the Philippines, Malaysia, Brunei, and Indonesia. The tarsier lives in the forests, leaping from branch to branch with extreme agility and speed.

A small animal, the tarsier's body is only 6 inches long, followed by a 10-inch tufted tail. It is covered in silky brown or gray fur and has a round face and huge eyes. Tarsiers have the largest eyes relative to body size of any mammal. Each eyeball is around 16 millimeters in diameter—the same size as its brain. The eyes are so large that they can't rotate, but a tarsier can twist its neck a full 180 degrees in either direction, like an owl. Its arms and legs are long and slender, as are its digits, which are tipped with rounded, fleshy pads to improve its grip on trees. Tarsiers are active

only at night, and spend their days hiding in tangles of vines or in the tops of tall trees. They are entirely carnivorous and subsist mainly on insects, reptiles, birds, and even bats. Although very curious animals, they tend to be loners.

Tarsiers have such specific requirements for their habitat and diet that the majority do not survive being put into captivity, which makes captive breeding programs nearly impossible to implement. With their populations declining across the board due to loss of habitat from agriculture, hunting, and logging, most species of tarsier are listed as vulnerable on the IUCN's Red List. The Siau Island Tarsier in particular is considered critically endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

Color illustration by Karen Montgomery, based on a black and white engraving from Lydekker's *Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.