Thinking in Vim

A Structural Reference to Modal Editing

Flyxion
Independent Researcher

2026

Preface

Most books about Vim are organized as inventories. They list commands, group them loosely by function, and trust the reader to assemble a working mental model from the accumulation of examples. This approach produces competent users of individual commands and, too often, users who never notice that the commands were never independent facts to begin with. They were instances of a small number of rules, applied over and over, in different combinations, against different kinds of text.

This book takes the opposite approach. It begins not with commands but with the entities Vim assumes exist: buffers, windows, registers, marks, modes, and the regions of text those modes and marks can address. It then derives the grammar that lets those entities combine, and only after the grammar is established does it turn to the specific transformations, that grammar makes possible. The intent is that a reader who finishes Part II should already be able to predict commands they have never seen, because they understand the rule that generates them rather than having memorized the rule's outputs.

The organizing progression is Ontology, Grammar, Geometry, Transformation, State, Automation, Application, and Reference. Each part depends on the one before it. Coordinate systems (Geometry) presuppose that the reader already knows what a motion is (Grammar), which presupposes knowing what a buffer is (Ontology). Automation presupposes a working vocabulary of transformations. Nothing in the main text is introduced before the concept it depends on has been established, with one deliberate exception: the encyclopedic appendices at the end, which are reference material rather than argument, and which the reader is free to consult out of order at any point.

This book is also deliberately conservative about scope. It covers core Vim: the editor as it has existed, largely unchanged in its fundamentals, for decades. Plugin ecosystems rise and fall; the operator-motion grammar, the register model, and the Ex language have not meaningfully changed since long before most current plugins existed. Neovim and its Lua configuration layer are mentioned where they diverge from core Vim in ways a reader should know about, but they are treated as an optional extension of the material here, not a parallel subject requiring separate treatment. A reader who learns core Vim from this book will be able to pick up Neovim, or the next editor built on the same ideas, without relearning the underlying grammar.

The practical workflows referenced throughout, particularly in the chapters on the Ex language and on Unix integration, are drawn from ordinary daily use rather than invented for demonstration: cleaning up manuscript text, editing configuration files, working inside Git, generating repetitive content, and the like. Vim is treated here not as an application with a feature list but as a text-processing language that happens to have a screen attached to it, one that composes naturally with the rest of the Unix toolchain rather than trying to replace it.

Anyone can read this book from the beginning and learn Vim from first principles.

Anyone who already knows Vim reasonably well can instead read Part II and the early chapters of Part V, where the underlying grammar is made explicit, and will likely find that commands they already use unreflectively suddenly make more sense as instances of a rule rather than as isolated facts to be recalled individually.

Contents

Preface **iii**

I Foundations **1**

1 What Makes Vim Different? **3**
1.1 Editors Versus Editing Languages . . . 3
1.2 The History of Vi and Vim . . . 3
1.3 Modal Editing . . . 4
1.4 Why Commands Compose . . . 5
1.5 Why Keyboards Outperform Menus . . . 5
1.6 Text as Structure Rather Than Pixels . . . 6

2 The Ontology of Vim **9**
2.1 Buffers . . . 9
2.2 Windows . . . 10
2.3 Tabs . . . 10
2.4 Files . . . 10
2.5 Registers . . . 11
2.6 Marks . . . 11
2.7 Text Objects . . . 11
2.8 Modes . . . 12
2.9 Commands . . . 12
2.10 The Help System . . . 13
2.11 Summary . . . 13

3 Installing and Configuring Vim **15**
3.1 Terminal Vim . . . 15
3.2 Terminal Versus GUI . . . 15
3.3 Vim in Git Bash . . . 16
3.4 Vim over SSH . . . 16
3.5 Your First vimrc . . . 17
3.6 Cursor Control Commands in vi . . . 18
3.7 Recovering From Mistakes . . . 18

II The Grammar of Editing **19**

4 The Language of Commands **21**
4.1 The Central Rule . . . 21
4.2 Counts . . . 21
4.3 Motions . . . 22
4.4 Operators . . . 22
4.4.1 Doubling . . . 23
4.5 Text Objects as Targets . . . 23
4.6 Repetition . . . 23
4.7 Composition and the Arithmetic of Coverage . . . 24

5 Coordinate Systems **25**
5.1 Character Coordinates . . . 25
5.2 Line Coordinates . . . 25
5.3 Word Coordinates . . . 26
5.4 Sentence and Paragraph Coordinates . . . 26
5.5 Section Coordinates . . . 26
5.6 Search Coordinates . . . 27
5.7 Screen Coordinates . . . 27
5.8 Absolute Coordinates . . . 27
5.9 Marks as a Coordinate System . . . 27
5.10 Choosing a Coordinate System . . . 28

6 Structural Regions **29**
6.1 The Inner/Around Distinction . . . 29
6.2 Word Regions . . . 29
6.3 Quote Regions . . . 30
6.4 Bracket and Parenthesis Regions . . . 30
6.5 Tag Regions . . . 30
6.6 Sentence and Paragraph Regions . . . 30
6.7 Block Regions . . . 31
6.8 User-Defined Regions . . . 31
6.9 Structural Regions and the Grammar . . . 31

III Transformations **33**

7 Operators **35**
7.1 Deletion . . . 35
7.2 Change . . . 35
7.3 Yanking . . . 36
7.4 Case Conversion . . . 36
7.5 Indentation . . . 36
7.6 Joining . . . 37
7.7 Filtering Through the Shell . . . 37

7.8 Formatting . . . 37
7.9 Operators as a Closed but Extensible Class . . . 37

8 Inserting and Replacing 39
8.1 Insert and Append . . . 39
8.2 Opening Lines . . . 39
8.3 Replace Mode . . . 39
8.3.1 Virtual Replace . . . 40
8.4 Substituting . . . 40
8.5 Completion . . . 40
8.6 Digraphs and Special Characters . . . 40
8.7 Insert Mode Is Still Insert Mode . . . 41

9 Undo and Time 43
9.1 Basic Undo and Redo . . . 43
9.2 The Undo Tree . . . 43
9.3 Persistent Undo . . . 44
9.4 Swap Files and Crash Recovery . . . 44
9.5 Undo as Confidence . . . 45

IV State 47

10 Registers 49
10.1 The Unnamed Register . . . 49
10.2 Named Registers . . . 49
10.3 Numbered Registers . . . 50
10.4 The Small Delete Register . . . 50
10.5 Read-Only Registers . . . 50
10.6 The Expression Register . . . 51
10.7 Clipboard Registers . . . 51
10.8 The Black Hole Register . . . 51
10.9 Registers as a System, Not a List . . . 51

11 Marks 53
11.1 Local Marks . . . 53
11.2 Global Marks . . . 53
11.3 Automatic Marks . . . 53
11.4 The Jump List . . . 54
11.5 The Change List . . . 54
11.6 The Alternate File . . . 54
11.7 Marks as Durable Coordinates . . . 55

12 Buffers, Windows, Tabs, Sessions **57**
12.1 Managing Buffers . . . 57
12.2 Splitting Windows . . . 57
12.3 Resizing and Rearranging . . . 58
12.4 Tabs . . . 58
12.5 Sessions . . . 58
12.6 Buffers, Windows, Tabs, and Sessions as a Single Hierarchy . . . 59

V Ex **61**

13 Ex as a Language **63**
13.1 The Basic Form . . . 63
13.2 Addresses . . . 63
13.3 Ranges . . . 64
13.4 Patterns as Addresses, Revisited . . . 64
13.5 Command-Line History . . . 65
13.6 Command-Line Completion . . . 65
13.7 Ex and Normal Mode as One Grammar . . . 65

14 Substitution **67**
14.1 Basic Substitution . . . 67
14.2 Flags . . . 67
14.3 Capture Groups . . . 68
14.4 Very Magic Mode . . . 68
14.5 The Replacement Expression . . . 68
14.6 Worked Examples . . . 68
14.7 Substitution as the Ex Language's Center of Gravity . . . 69

15 Global Editing **71**
15.1 The Global Command . . . 71
15.2 The Inverse Global Command . . . 71
15.3 Worked Examples . . . 71
15.4 The `:normal` Command on Its Own . . . 72
15.5 The `:execute` Command . . . 72
15.6 Buffer, Window, and Argument List Iteration . . . 73
15.7 Global Editing as the Ex Language Completed . . . 73

VI Automation **75**

16 Macros **77**
16.1 Recording . . . 77
16.2 Testing Before Scaling . . . 77
16.3 Nested and Recursive Macros . . . 78
16.4 Defining a Macro Without Recording It . . . 78

16.5 Editing a Recorded Macro . . . 79
16.6 The Pain-Point Threshold . . . 79
16.7 Macro Libraries . . . 80

17 Vimscript **81**
17.1 Variables . . . 81
17.2 Functions . . . 81
17.3 Conditionals . . . 82
17.4 Loops . . . 82
17.5 Mappings . . . 83
17.6 Autocommands . . . 83
17.7 Vimscript as the Ceiling of This Book's Automation Material . . . 83

VII Unix Integration **85**

18 Shell Commands **87**
18.1 Executing a Command . . . 87
18.2 Reading Command Output Into the Buffer . . . 87
18.3 Filtering the Buffer Through a Command . . . 87
18.4 Sending the Buffer to a Command . . . 88
18.5 Working Directory and Sub-Shells . . . 88
18.6 A Note on Workflow . . . 88
18.7 Shell Integration as a Pipeline Stage . . . 89

19 Development Workflows **91**
19.1 Commit Messages . . . 91
19.2 Interactive Rebase . . . 91
19.3 Merge Conflicts . . . 92
19.4 Comparing Files with `vimdiff` . . . 92
19.5 The Quickfix List and `make` . . . 93
19.6 Searching a Project with `grep` . . . 93
19.7 Editing History as Editing Text . . . 93

20 Writing **95**
20.1 Markdown . . . 95
20.2 LaTeX . . . 95
20.3 Large Manuscripts . . . 96
20.4 Notes and Documentation . . . 96
20.5 Writing as the Terminus of This Book's Argument . . . 97

VIII Reference **99**

Using This Reference **101**

A Every Normal-Mode Command **103**
A.1 Character and Line Movement . . . 103
A.2 Word, Sentence, and Paragraph Movement . . . 103
A.3 Search and Character-Find Movement . . . 104
A.4 Screen and Absolute Position . . . 104
A.5 Operators . . . 104
A.6 Inserting and Replacing . . . 105
A.7 Deletion and Change Shorthand . . . 105
A.8 Put, Undo, and Repeat . . . 105
A.9 Marks and Registers . . . 106
A.10 Macros . . . 106
A.11 Visual Mode . . . 106
A.12 Windows and Tabs . . . 107
A.13 Miscellaneous . . . 107

B Every Ex Command **109**
B.1 File and Session Management . . . 109
B.2 Buffers, Windows, and Tabs . . . 109
B.3 Substitution and Pattern Commands . . . 110
B.4 Execution and Automation . . . 110
B.5 Shell Interaction . . . 110
B.6 Build, Search, and Quickfix . . . 111
B.7 History and Inspection . . . 111
B.8 Options . . . 111

C Every Text Object **113**

D Every Register **115**

E Every Option **117**

F Every Built-in Function **119**

G Regular Expression Reference **121**
G.1 Anchors and Classes . . . 121
G.2 Quantifiers and Grouping . . . 121
G.3 Replacement Syntax . . . 122

H Command-Line Arguments **123**

I vimrc Cookbook **125**

J Historical Evolution of Vim **127**

Part I

Foundations

Chapter 1

What Makes Vim Different?

1.1 Editors Versus Editing Languages

Most text editors present themselves as applications with a feature list: a set of menus, a set of keyboard shortcuts bound to those menu items, and a canvas on which text appears and can be manipulated more or less as it would be manipulated on paper. The editor is a tool, and the user operates it directly, the way one operates a typewriter.

Vim is better understood as a language with an editor attached to it. Its commands are not a flat list of shortcuts bound to actions; they are utterances in a small grammar, composed of a countable number of primitive parts (operators, motions, text objects, registers) that combine according to a fixed set of rules to produce an effectively unbounded number of distinct edits. Learning Vim is less like memorizing a control panel and more like learning the morphology of a language: once the rules for combining a small vocabulary are understood, fluency in producing new, never-previously-seen combinations follows naturally, in the same way a competent speaker of a language can produce and understand sentences they have never encountered before.

This distinction matters because it changes what "learning Vim" means. A user who learns Vim as a list of shortcuts eventually plateaus: they know the commands they have looked up, and unfamiliar text-editing problems require a new search. A user who learns Vim as a grammar does not plateau in the same way, because an unfamiliar problem is usually just an unfamiliar combination of familiar primitives. The rest of this book is organized around making that grammar explicit from the outset, rather than letting it remain implicit and be discovered gradually, command by command, over years of use.

1.2 The History of Vi and Vim

Vim's design did not appear from nothing; it is the current endpoint of a lineage that runs back through vi to the line editor ed, and understanding that lineage explains several design decisions that otherwise look arbitrary.

ed, developed in the early 1970s, was a line editor: it operated on one line of a file at a time, addressed by line number or by a search pattern, with no visual display of surrounding context beyond what the user explicitly requested. Its command language, terse by necessity given the teletype hardware of the era, is the direct ancestor of what is now called Ex mode in Vim: commands like `:s` for substitution, `:g` for running a command on every line matching a pattern, and address ranges like `1,10` or `%` all originate here, largely

unchanged in syntax.

vi, written by Bill Joy in the late 1970s, added a visual, full-screen mode on top of ex's line-oriented command language, taking advantage of the video terminals that had become available. Critically, vi did not discard the ex command language; it added a second mode of interaction (screen-oriented, character-addressable movement and editing) alongside the first. This is the origin of vi's, and later Vim's, dual nature: a full-screen editor for direct manipulation of visible text, sitting on top of a line-oriented command language for operations that are more naturally expressed as text than as gestures. A reader who has ever wondered why Vim has both `dd` (a screen-oriented command) and `:d` (a line-oriented command) that accomplish related things is seeing this history directly.

Vim itself (the name originally stood for "Vi IMitation," later reinterpreted as "Vi IMproved") was written by Bram Moolenaar beginning in 1988, initially for the Amiga, as a vi-compatible clone extended with features vi lacked: multi-level undo, syntax highlighting, split windows, a scripting language, and eventually a large ecosystem of built-in and pluggable extensions. Vim preserved vi's command grammar essentially intact while extending it, which is why nearly everything in this book that traces back to classic vi still works, unmodified, in current Vim.

Neovim, forked from Vim in 2014, preserves this grammar as well, while modernizing the implementation (a different plugin architecture, first-class Lua scripting alongside Vimscript, and a design oriented toward embedding the editor inside other tools). For the purposes of this book, the distinction rarely matters: the grammar of operators, motions, text objects, registers, and the Ex language is shared between Vim and Neovim, and material that depends on one or the other is flagged explicitly where it appears.

1.3 Modal Editing

The single design decision that most distinguishes vi and Vim from nearly every other text editor is modal editing: the same keys mean different things depending on which mode the editor is currently in. In Normal mode, the letter `i` inserts text; in Insert mode, it types the letter i. This is initially disorienting to users accustomed to non-modal editors, where a keystroke always produces the same result regardless of context, and it is the single most common source of early frustration with Vim.

The rationale for modality becomes clear once the alternative is examined closely. In a non-modal editor, every key that could plausibly be part of typed text (nearly the entire keyboard) is unavailable for use as a command, because pressing it must insert that character. Commands are therefore relegated to key combinations involving modifier keys (Control, Alt, or Command), menu selections, or multi-key mnemonic sequences, none of which are as fast to press as a single unmodified key. Vim's modal design inverts this: in Normal mode, the entire alphabet, along with most punctuation, is available as a one-keystroke command, because no character is being inserted in that mode. The cost of modality is that the user must track which mode they are in; the benefit is that the entire keyboard becomes command space rather than being reserved for text entry, which is what makes the compact, high-density command grammar described in the rest of this book possible in the first place.

Vim extends this basic Normal/Insert distinction with several further modes, each addressing a different category of operation: Visual mode for selecting a region before acting on it, Command-line mode for entering Ex commands, Replace mode for overtyping existing text, and others enumerated in Chapter 2 and taken up in detail as each becomes relevant later in the book. Each mode exists because it changes what the same small set of keys means, and that reuse of keys across modes, rather than a proliferation of modifier-key combinations, is precisely what keeps the command surface learnable.

1.4 Why Commands Compose

A non-modal editor with a menu-driven feature set tends to grow its command surface by addition: each new capability gets a new menu item, a new keyboard shortcut, or a new dialog box, and the total number of things a user must separately learn grows roughly linearly with the number of features. Vim's command surface grows differently, because most of its commands are not atomic features but products of composition.

Consider deletion. Vim does not have a separate command for "delete a word," "delete to the end of the line," "delete a sentence," and "delete a paragraph." It has one operator, `d`, and a set of independently useful motions (`w` for a word, `$` for the end of the line, `)` for a sentence, `}` for a paragraph), each of which is also usable on its own for movement. The deletion commands are not memorized as separate facts; they are the operator applied to motions the user already knows for an entirely different purpose. The same is true of changing text (`c`), yanking it into a register (`y`), and every other operator introduced in Part III: each one multiplies against the full set of motions and text objects, so that learning one new operator effectively adds dozens of usable commands at once, at zero additional memorization cost for the motions themselves.

This compositional structure is the mechanism, not merely the metaphor, behind treating Vim as a language rather than a feature list. Part II makes this rule explicit as `[count] operator motion`, and the rest of the book is largely an exploration of what can fill each of those three slots.

1.5 Why Keyboards Outperform Menus

The case for a keyboard-centric editing language over a menu-and-mouse interface is not aesthetic; it rests on a specific, measurable claim about the physical cost of switching input modalities. Moving a hand from the home row to a mouse, acquiring a target with the pointer, clicking, and returning the hand to the keyboard takes substantially longer than any single keystroke, and that cost is paid every time the interaction happens, regardless of how visually intuitive the menu item was to locate. A command that can be issued without leaving the home row is not merely faster to execute once fluency is achieved; it removes an entire category of physical transition that a mouse-mediated command cannot avoid.

This argument does not depend on typing speed in the conventional sense (words per minute of prose entry) but on the frequency and cost of small, repeated editing operations: moving to a specific character, deleting a word, jumping to a matching bracket, repeating the last change. These are the operations a programmer, a writer editing a long manuscript,

or anyone doing sustained textual work performs dozens or hundreds of times per session, and it is precisely at that frequency that even a small per-operation cost compounds into a substantial difference in total friction. A command that takes a mouse user a second and a half to execute, most of it spent moving the hand and acquiring a target, might take a fluent Vim user under a quarter of a second, and the difference is felt not as raw speed but as a kind of continuity: the editing process does not keep interrupting itself to ask the hand to travel somewhere else.

None of this is an argument that graphical interfaces are worthless, only that for the specific, high-frequency, small-grained operations that dominate serious text editing, a compact command language operable without leaving the home row has a structural advantage that no amount of visual polish in a menu system can offset.

The strongest evidence for this argument is not anything Vim's own documentation claims about itself but what its users do once they have internalized it. It is not unusual for a fluent Vim user to eventually remap the arrow keys in unrelated software to `hjkl`, or to rebind a completely unrelated application's movement controls to the same four keys, not because the software in question has any connection to Vim, but because the underlying ergonomic argument, home row over hand travel, applies just as well to any application with directional controls as it does to a text buffer. The same reasoning that motivates learning Vim in the first place tends to generalize into a broader habit: any command that is used often enough, or is annoying enough to keep looking up, becomes a candidate for a permanent keyboard shortcut, wherever it happens to live. System-wide remapping tools such as AutoHotkey on Windows make this habit practical to extend well beyond any single application, to the point that a consistent, personal command layer can sit on top of an entire operating system rather than being confined to one editor.

This generalizing impulse is worth taking seriously as a claim about what Vim actually teaches. A reader who finishes this book having memorized its commands has learned an editor. A reader who finishes it having internalized the underlying principle, that repeated, looked-up, or annoying actions belong on the keyboard rather than in a menu, has learned something that keeps paying off long after any specific command has been forgotten and relearned from the help system. Chapter 4 returns to this principle in a more structural form once the operator-motion grammar has been introduced; the point to take from this chapter is only that the habit is real, widely reported among fluent users, and not an eccentricity specific to text editing.

1.6 Text as Structure Rather Than Pixels

The final distinguishing property of Vim, and the one most directly responsible for the shape of Parts II through IV of this book, is that Vim treats the file being edited as structured text rather than as an array of pixels or an opaque visual canvas. A word is a recognized unit with a defined boundary, computed from the buffer's content and the current `iskeyword` settings, not a region the user has to select by dragging. A sentence, a paragraph, a matching pair of brackets, and a quoted string are each addressable as units for exactly the same reason: Vim parses enough of the buffer's structure, at edit time, to know where such a unit begins and ends, and exposes that knowledge as a target for commands.

This is the deep reason text objects (Chapter 10) are able to exist at all, and it is worth stating plainly this early, because it recurs throughout the book: every time a later chapter introduces a new kind of region a command can act on, from Vim's own most primitive character motion up through user-defined text objects, the underlying claim is the same one made here, that text in a Vim buffer is never merely a rectangle of pixels but is always addressed as a member of some class of structured object, and that class membership, rather than screen position, is what most editing commands actually operate on.

A GUI editor that treats text as pixels on a canvas can still support word selection, sentence selection, and so on, typically by double-clicking or triple-clicking to invoke a hard-coded selection heuristic. The difference in Vim is that this structural awareness is not a handful of special-cased mouse gestures bolted onto an otherwise pixel-oriented model; it is the default and general way the editor understands its own content, available uniformly to every operator, from every mode, without special-casing. That generality, more than any individual feature, is what the rest of this book sets out to demonstrate.

Chapter 2

The Ontology of Vim

Before any grammar can be introduced, its terms have to be named. A sentence about deleting a word presupposes that both ”deletion” and ”word” already denote something in particular; a command like `daw` is meaningless until the reader knows what a text object is, just as much as it is meaningless until they know what an operator is. This chapter is that naming. It does not yet explain how these entities combine (that is the task of Part II) or what can be done with them (Part III onward). It only establishes what exists: the vocabulary the rest of the book will assume without re-explaining.

The entities below are not of the same kind. Some are persistent containers that outlive any single edit (buffers, registers). Some are transient views onto those containers (windows, tabs). Some are addresses into text (marks) or classes of region within it (text objects). Some are not things at all but states the editor itself can be in (modes). Keeping these categories distinct from the outset avoids a common source of confusion for newcomers, who often conflate a buffer with the window that happens to be displaying it, or a mark with the text object it can help construct.

2.1 Buffers

A buffer is Vim’s in-memory representation of a file’s content. When a file is opened, its contents are read into a buffer; edits are made to that buffer, not to the file on disk, until the buffer is explicitly written back with a command like `:w`. This distinction, buffer versus file, is not a technicality. A buffer can exist with no corresponding file at all (an empty scratch buffer created with `:enew`), and a single file can be associated with only one buffer at a time even if that buffer is currently displayed in several windows simultaneously. Every open file corresponds to exactly one buffer; every buffer does not necessarily correspond to a file.

Buffers persist independently of whether they are currently visible. A buffer can be ”hidden,” meaning it retains its content and any unsaved changes while no window is currently displaying it. This is what makes it possible to have dozens of files open in a single Vim session while only ever looking at one or two of them at a time: the buffer list is a superset of what is presently on screen. Chapter 12 covers the commands for listing, switching between, and managing buffers directly; for now, the essential fact is that a buffer is Vim’s unit of ”a file’s content, currently loaded,” and it is the thing every other entity in this chapter ultimately operates on or displays.

2.2 Windows

A window is a viewport onto a buffer. It occupies a rectangular region of the screen, has its own cursor position, its own scroll position, and its own local settings for certain options, but the buffer it displays is shared state: editing text through one window immediately changes what any other window displaying the same buffer shows, because there is only one buffer, not one per window. Vim allows the screen to be divided into multiple windows, arranged by horizontal or vertical splits (Chapter 22), which is what makes it possible to view two different regions of the same file, or two entirely different files, side by side.

The buffer/window distinction is the single most useful conceptual separation in this chapter for a newcomer to internalize early, because it explains behavior that otherwise looks surprising: closing a window with `:q` does not delete the buffer it was displaying, provided that buffer is still open in another window or has unsaved changes Vim is unwilling to discard; and opening the same file in two separate windows does not produce two independent copies of it to reconcile later, because both windows are looking at the same underlying buffer the entire time.

2.3 Tabs

A tab page, confusingly for anyone arriving from a browser's notion of "tabs," is not a container for a single file. It is a container for an arrangement of windows: a tab page can itself be split into several windows, each showing its own buffer. Switching tabs with `gt` or `gT` does not switch which file is open; it switches which saved window layout is currently visible. A user who wants to keep one project's files in one arrangement of splits and a second project's files in a completely different arrangement, and to switch between those two arrangements as a unit, is the intended use case for tabs. Chapter 23 covers this in more depth, including how tabs relate to sessions, which allow an entire multi-tab, multi-window, multi-buffer layout to be saved to disk and restored later.

2.4 Files

The relationship between a file (a named object on disk) and a buffer (Vim's loaded representation of that file's content) has already been introduced above, but it is worth stating the asymmetry explicitly, because several later chapters depend on it. A buffer can be modified without the underlying file changing at all, for as long as the user chooses not to write it. A file can change on disk, by some other process entirely, while a buffer holding a now-stale copy of its old content remains open and blissfully unaware, until Vim is prompted to check. Vim is, at its core, an editor of buffers that happens to load them from and save them to files; it is not, at the moment-to-moment level of an editing session, an editor of files directly. This distinction is also what makes constructs like scratch buffers, and Vim's ability to read a shell command's output directly into a buffer without ever touching disk (`:r !command`), coherent rather than exceptional: a buffer never actually required a file to exist in the first place.

2.5 Registers

A register is a small, named piece of persistent storage, holding a span of text, that survives across edits until it is next overwritten. Every deletion, change, and yank in Vim writes to some register, whether the user names one explicitly or not; the unnamed register (accessed as `"`) is what an unadorned `p` or `P` reads from, and it is silently updated by nearly every deleting or yanking command. Named registers, addressed with a letter (`"a` through `"z`), give the user a way to hold more than one piece of text at once, deliberately, rather than being limited to whatever the most recent operation happened to leave behind.

Registers are not merely a clipboard, and treating them as one obscures most of what makes them useful. There are numbered registers that automatically cycle through recent deletions, letting older deleted text be retrieved even after several more deletions have occurred; a read-only register holding the contents of the most recent search pattern; an expression register that evaluates arbitrary Vimscript and inserts the result; and, on systems where Vim is compiled with clipboard support, registers that read from and write to the operating system's own clipboard, bridging Vim's internal storage with the rest of the desktop environment. Chapter 16 treats every register in detail; the point to establish here is only that "register" denotes a general mechanism for named, persistent text storage, of which the familiar copy-paste clipboard is one instance rather than the whole story.

2.6 Marks

A mark is a saved position within a buffer: a specific line and column, given a one-character name, to which the cursor can later be returned directly. Local marks, named with a lowercase letter (`ma` sets mark `a`), are specific to the buffer in which they were set. Global marks, named with an uppercase letter, are specific to a position in a particular file but are addressable from any buffer, which makes them the mechanism for jumping directly to a remembered location in an entirely different file without first having to open and navigate to it manually.

Marks matter to this book for a reason beyond simple navigation: they are one of the things a motion can target, which means marks are among the building blocks operators compose with. The command `d'a` deletes from the cursor to mark `a`, treating the mark as a motion target in exactly the same grammatical slot a word or a search pattern would occupy. This is examined properly once the operator-motion grammar is introduced in Part II; here it is enough to know that a mark is a named, addressable position, and that Vim also maintains several marks automatically (the position before the last jump, the boundaries of the last changed or yanked text) without the user ever having to set them by hand, covered fully in the chapter on Marks in Part IV.

2.7 Text Objects

A text object denotes a structured region of text, defined not by explicit start and end positions the user specifies but by the syntactic or semantic unit it belongs to: a word, a quoted string, the contents of a pair of parentheses, a paragraph, a sentence. Where a motion moves

the cursor to a position, a text object identifies a span, generally without moving the cursor itself when used as the argument to a command; `iw` does not mean "go to the word," it means "the inner word, wherever the cursor currently sits within it."

Text objects come in two forms for most of the structures they describe: an inner form (`i`) that selects the content of the structure without its delimiters, and an around form (`a`) that includes them. `ci"` changes the text inside a pair of double quotes without touching the quote marks themselves; `ca"` changes the quoted text and the quote marks together. This inner/around distinction recurs across nearly every text object Vim defines and is worth internalizing as a general pattern rather than memorizing separately for each one. Chapter 10 catalogs the full set; this section's purpose is only to place "text object" in the ontology as a distinct kind of thing from a motion, a mark, or a register, because later chapters will depend on the reader not conflating the three.

2.8 Modes

A mode is a state the editor itself is in, which determines what the same keystroke means. This was introduced conceptually in Chapter 1; here it is enough to enumerate what modes exist, without yet explaining their internal behavior in full, which is instead taken up as each mode becomes relevant to the surrounding material: Insert and Replace mode in the next part's chapter on Inserting and Replacing, Visual mode as it becomes relevant to later transformations, and Command-line mode throughout Part V. Vim distinguishes at minimum Normal mode (the default, for issuing commands), Insert mode (for typing text directly into the buffer), Visual mode (for selecting a region before acting on it, itself subdivided into character-wise, line-wise, and block-wise variants), Command-line mode (for entering Ex commands, searches, and filter expressions), and Replace mode (for overtyping existing characters rather than inserting new ones ahead of them). Terminal mode, present in Vim versions with an embedded terminal emulator, is a further mode specific to that feature. What unifies all of these, and what justifies treating "mode" as a first-class entity in this ontology rather than an implementation detail, is that the meaning of every subsequent keystroke in this book is conditioned on which mode is active at the moment it is pressed, and a great deal of Vim's apparent inconsistency to new users evaporates once this conditioning is made explicit rather than left implicit.

2.9 Commands

A command, in the broadest sense used in this book, is any complete instruction issued to Vim, whether it is a single Normal-mode keystroke, a composed sequence like `daw`, or a line typed in Command-line mode beginning with a colon. This is a deliberately broad category, and it is broad on purpose: one of the central claims of Part II is that Normal-mode key sequences and Ex commands are not two unrelated command languages coexisting inside one editor, but two surfaces of the same underlying grammar, addressed differently (one by direct keystroke, one by typed line) but sharing much of the same vocabulary of ranges, patterns, and targets. This chapter does not yet defend that claim; it only flags, as part of

naming what exists, that "command" is being used here to cover both surfaces rather than being reserved for one of them.

2.10 The Help System

Vim ships with an extensive, cross-referenced help system, accessed with `:help` (or its abbreviation `:h`), covering every command, option, and feature described anywhere in this book, and a great many that are not. It is included here in the ontology chapter, rather than left as an afterthought, because it is not merely a manual bolted onto the editor; it is itself a Vim buffer, opened in a split window, navigable with the same motions, searches, and jumps as any other buffer, and containing hyperlink-like tags (rendered distinctly, and followed with `Ctrl-]`) that let the reader jump directly from a mention of a related command to that command's own help entry. A reader who has internalized nothing else from this chapter except how to move `:help` out of the way, follow a tag, and jump back with `Ctrl-o` already has a self-sufficient way to go beyond anything covered in this book, using the exact editing skills the rest of the book is teaching to navigate the documentation of the editor itself.

2.11 Summary

Nine entities have been named: buffers as the loaded content of a file, windows as viewports onto a buffer, tabs as saved arrangements of windows, files as the on-disk objects buffers are loaded from and written to, registers as named persistent text storage, marks as named positions, text objects as structured regions, modes as states that condition what a keystroke means, and commands as the general term for complete instructions in either the Normal-mode or Ex surface of Vim's grammar. None of these has yet been shown in combination with any other; that combination, and the rules that govern it, is the subject of Part II.

Chapter 3

Installing and Configuring Vim

The material in this chapter is the least conceptually interesting in the book and the most likely to go stale, since installation procedures and package names shift with operating system releases in a way the editing grammar covered in later chapters does not. It is included anyway, briefly, because a reader cannot practice anything in Part II without a working Vim in front of them, and because the choice between the environments described here (a terminal, a graphical build, a Git Bash shell, a remote server over SSH) is itself a useful early illustration of a theme that recurs throughout the book: the editing language stays constant even as the surface it runs inside changes.

3.1 Terminal Vim

On any Linux distribution, Vim is typically available directly through the system package manager: `apt install vim` on Debian and Ubuntu-derived systems, `dnf install vim` on Fedora, `pacman -S vim` on Arch. macOS ships a minimal build of vi by default, but a full-featured Vim is readily available through Homebrew (`brew install vim`). What matters more than the exact package name, which the reader's own system documentation will confirm faster than a book can, is the distinction between a minimal build and a full-featured one: many systems ship a stripped-down `vim-tiny` or equivalent, lacking scripting language support, clipboard integration, or other features this book assumes are present. Where a distribution offers both, the full build (often named `vim` as opposed to `vim-tiny`, or selected via `vim-gtk3` or similar on Debian-derived systems for clipboard support specifically) is the one this book assumes.

Once installed, `vim --version` reports which optional features were compiled in, listed with a leading + for enabled and − for disabled. A reader who finds a feature described later in this book not behaving as expected, particularly anything involving the system clipboard (Chapter 16) or Vimscript's more advanced functions (Chapter 24), should check this output before assuming the fault lies in their configuration rather than their build.

3.2 Terminal Versus GUI

Vim's terminal build and its graphical counterpart, gVim (invoked as `gvim` on most systems, or `mvim` on macOS via the MacVim project), share the same core editing grammar entirely; nothing in Parts II through VII of this book distinguishes between them. What differs is peripheral: gVim offers native menus and toolbars (which this book's plugin-

free, keyboard-first philosophy largely ignores, and which can be disabled through the `guioptions` setting introduced later in this chapter, for a leaner interface), font selection through the operating system's own font dialog or the `guifont` option, and, notably, more reliable access to the system clipboard without additional configuration, since a terminal-based Vim's access to the clipboard depends on how the terminal emulator itself is configured and on Vim having been compiled with clipboard support.

The practical recommendation, for a reader without a strong existing preference, is to begin in a terminal, since nearly every example in this book, and nearly every real workflow described in Part VII (editing over SSH, inside a Git rebase, from a shell script), assumes a terminal context, and a reader fluent only in gVim's menu-assisted environment will otherwise face a second, avoidable adjustment later.

3.3 Vim in Git Bash

Readers on Windows who have installed Git for Windows have Git Bash available, a Bash environment bundled with a terminal build of Vim, and this is frequently the first place a Windows user encounters Vim involuntarily rather than by choice: it is Git's default editor for commit messages, interactive rebases, and merge conflict resolution (a workflow covered in depth in Chapter 31). Git Bash's bundled Vim is a genuine, if sometimes older, build of the real editor, not an emulation, so everything in this book applies to it directly. The one adjustment worth flagging in advance is clipboard behavior: Git Bash's terminal and the Windows clipboard do not always interoperate as smoothly as a native Linux or macOS terminal does with its host system, which is worth knowing before Chapter 16's coverage of clipboard registers, so that unexpected behavior there is recognized as an environment quirk rather than a misunderstanding of the register model itself.

3.4 Vim over SSH

Editing a file on a remote machine by connecting over SSH and running a terminal-based Vim directly on that machine, rather than mounting the remote filesystem locally and editing with a GUI tool, is one of the more common real-world contexts in which Vim's terminal-only, keyboard-only design stops being a stylistic preference and becomes a practical necessity: a graphical editor simply is not available in that context, while a terminal-based Vim, needing only a text-mode connection, works identically to how it works locally. This is worth stating plainly early in the book, because it is one of the strongest practical answers to the question of why anyone would deliberately choose a keyboard-driven, splash-free editor in an era of graphical alternatives: sooner or later, most people who administer or deploy software end up needing to edit a file on a machine where nothing but a terminal is available, and Vim, learned once, is available there without modification.

Nothing about the editing grammar covered later in this book changes over an SSH connection. What can change, and is worth a brief mention here rather than a full treatment, is latency: on a slow or high-latency connection, visual feedback for every keystroke can lag noticeably, which makes commands that batch several edits into one action (macros, covered in Part VI, and Ex commands with ranges, covered in Part V) considerably more

pleasant to use than a long sequence of individual keystrokes, each waiting on a round trip to render.

3.5 Your First vimrc

Vim reads a configuration file at startup, conventionally named `.vimrc` in the user's home directory on Linux and macOS (`~/.vimrc`), or `_vimrc` on Windows, in which options can be set, key mappings defined, and Vimscript executed, all before the editor is ready for input. The path Vim expects can always be confirmed from inside a running session with `:echo $MYVIMRC`, and the file can be opened directly for editing with `:e $MYVIMRC`, then reloaded into the current session without restarting Vim with `:source $MYVIMRC`, a pairing worth committing to memory early, since configuring Vim is itself an iterative editing task best done inside Vim.

A minimal first vimrc, sufficient to follow the rest of this book comfortably, need not be more than a handful of lines:

```
set nocompatible
syntax on
set number
set incsearch
set ignorecase
set smartcase
set expandtab
set shiftwidth=4
set softtabstop=4
```

`set nocompatible` disables backward-compatibility behaviors inherited from classic vi that are rarely wanted in a modern Vim session (in current Vim this is frequently the default already, but stating it explicitly costs nothing and documents intent). `syntax on` enables syntax highlighting. `set number` displays line numbers in the left margin. `set incsearch` shows search matches as the pattern is typed, rather than only after pressing return. `set ignorecase` together with `set smartcase` makes searches case-insensitive by default but automatically case-sensitive the moment the search pattern itself contains an uppercase letter, a small combination that in practice does the right thing far more often than either setting alone. The final three lines configure indentation to use spaces rather than literal tab characters, at a width of four columns, which is a matter of preference rather than necessity and is included here only as a placeholder the reader is expected to adjust.

This is deliberately minimal. The chapter on Vimscript in Part VI returns to configuration in greater depth, covering mappings, autocommands, and the fuller range of options; the vimrc above is meant only to remove the friction of an unconfigured first session, not to represent a finished configuration. One further option is worth naming here specifically because it was promised earlier in this chapter: `set guioptions-=m` and `set guioptions-=T`, added to the vimrc on a system running gVim, remove the menu bar and toolbar respectively, which is the mechanism behind the leaner graphical interface mentioned in Section 3.2.

3.6 Cursor Control Commands in vi

Before configuration is set aside in favor of Part II's grammar, it is worth previewing the most basic motions, since every subsequent chapter assumes the reader can already move the cursor without conscious effort. In Normal mode, `h`, `j`, `k`, and `l` move the cursor left, down, up, and right respectively, one character or line at a time, arranged on the home row specifically so that the hand need not leave its resting position to navigate, the ergonomic argument made in Chapter 1 realized in its simplest possible form. `G` moves to the last line of the file; a count prefixed to `G` (`25G`) moves to that specific line number instead. These four letters and one command are not a complete account of movement, which is the subject of Part II, but they are enough to get a reader through this book's earliest examples without having to skip ahead.

3.7 Recovering From Mistakes

Two safety nets are worth knowing before anything else in this book is attempted. First, `u` undoes the most recent change, and can be pressed repeatedly to step back through the undo history; `Ctrl-r` redoes a change that was just undone. Second, `:q!` discards all unsaved changes to the current buffer and closes it without writing, which is the fastest way out of a file that has been edited into a state the reader would rather abandon than repair. Neither of these requires any grammar not already covered in this chapter, and both are worth using freely and without hesitation while working through the rest of this book: nothing here is dangerous to a file on disk unless `:w` is explicitly issued, and even changes that have been written can usually be recovered through the undo history covered fully in Chapter 13.

Part II

The Grammar of Editing

Chapter 4

The Language of Commands

Chapter 1 argued that Vim's commands are utterances in a grammar rather than entries in a list, and Chapter 2 named the entities that grammar operates on. This chapter makes the grammar itself explicit. Everything in Parts III through VII is, in one sense, an extended elaboration of the single rule stated here; the remainder of the book is occupied with filling in what can appear in each of this rule's positions; and a reader who has genuinely absorbed this chapter, rather than skimmed it, will find that a great deal of what follows reads less like new material and more like the expected consequence of a rule already understood.

4.1 The Central Rule

The core of Normal-mode editing in Vim can be written as:

```
[count] operator motion
```

or, when the target is a text object rather than a motion:

```
[count] operator text-object
```

An operator is a command that does not act by itself; it waits for something to act upon. The letter `d`, pressed alone, does nothing but wait. `dw` completes the sentence: the operator `d` (delete) applied to the motion `w` (to the beginning of the next word), meaning "delete from the cursor to the beginning of the next word." The count is optional and, when present, is most often applied to the motion rather than the operator: `d3w` deletes through three words, not "delete three times." Section 4.2 returns to this distinction in more depth, since count placement is one of the few places the grammar has a genuine ambiguity worth resolving explicitly.

This single rule, once internalized, replaces what would otherwise be a combinatorial explosion of commands to memorize individually. Section 4.7 works through the arithmetic of that claim directly.

4.2 Counts

A count is a number typed before a command, and it means "repeat this command *n* times," with the specific effect of that repetition depending on what kind of command it prefixes. Before a motion used alone, `5j` moves the cursor down five lines rather than one. Before

an operator-motion pair, the count can be placed on either the operator or the motion, and, for most operator-motion combinations, the two placements are equivalent: `2dw` and `d2w` both delete two words, because Vim multiplies any count on the operator together with any count on the motion before applying either. `3d2w` deletes six words, the product of both counts, which is worth knowing exists even though it is rarely used deliberately; what matters in practice is that a count typed before either the operator or the motion in a pair will generally do what its plain-language reading suggests.

Where this can mislead a reader is in commands that are not, despite appearances, an operator-motion pair at all. `5dd` deletes five lines, not because the count multiplies onto a motion, but because `dd` is itself a special, doubled form (Section 4.4.1) that the count modifies as a unit. The rule to hold onto is not "counts always multiply the same way" but "a count means *n* repetitions of whatever the following command, taken as a whole, does," which happens to look like simple multiplication for genuine operator-motion pairs and looks like something else for the handful of specialized doubled and Ex forms encountered later.

4.3 Motions

A motion is a command that moves the cursor, and Vim's central architectural decision, already anticipated in Chapter 1, is that motions are not a separate category from movement commands the user issues on their own; they are the same commands, reused. Pressing `w` in isolation moves the cursor to the beginning of the next word; the very same `w`, following an operator, tells that operator where its target ends. Nothing about `w` changes between these two uses; what changes is only whether an operator was waiting for it.

This reuse is what makes the grammar generative rather than merely compact. Chapter 5 catalogs Vim's motions properly, organized by the coordinate system each one navigates; the point to establish here is only the mechanical fact that any motion, once learned for its own sake as a way of moving the cursor, is simultaneously and automatically learned as a valid target for every operator in the language, at no additional cost.

4.4 Operators

An operator is a command that takes a motion or text object as its argument and performs some transformation on the span between the cursor's starting position and wherever that motion or text object indicates. Vim defines a modest number of core operators: `d` (delete), `c` (change, meaning delete and enter Insert mode at that point), `y` (yank, meaning copy into a register without removing the original), and a smaller number of formatting operators such as `gU` (uppercase) and `=` (auto-indent), covered in full in the next chapter. The core claim of this section is that the number of core operators is deliberately small, because each one is designed to combine with the entire motion and text-object vocabulary rather than to exist as a narrow, special-purpose command.

Operators can themselves be doubled to act on the current line as a unit, discussed next, and, as Chapter 6 develops at length, can take a text object rather than a motion as their target, which is where the grammar's descriptive power becomes most apparent: an operator

paired with a semantic region, rather than a cursor movement, produces commands like `ci"` (change inside quotes) that would be nearly impossible to state cleanly as a single dedicated command but fall out immediately once operator and text object are understood as independently combinable parts.

4.4.1 Doubling

Several operators have a doubled form, pressing the same letter twice, that acts on the current line as a whole rather than requiring an explicit motion: `dd` deletes the current line, `yy` yanks it, `cc` changes it. This is not a separate grammatical rule so much as a convenient shorthand for "this operator applied to the whole-line motion," and it exists because acting on entire lines is common enough to deserve a two-keystroke idiom rather than requiring the user to spell out a line-wise motion explicitly every time.

4.5 Text Objects as Targets

Where a motion describes a path the cursor could travel, a text object (introduced in Chapter 2) describes a region directly, generally without implying any particular direction of travel to get there. This distinction matters grammatically: `daw` does not mean "delete while moving to the word," it means "delete the region that constitutes a word, including its surrounding whitespace." The rule from Section 1 accommodates this directly, since a text object simply fills the same grammatical slot a motion would, differing only in what kind of span it hands to the operator: a path-derived span for a motion, a structurally-defined span for a text object. Chapter 6 treats the full catalog of text objects and their inner/around distinction; here the point is only to confirm that they occupy the same slot in the sentence as a motion, which is why the same small set of operators works identically whether it is given a motion or a text object to act on.

4.6 Repetition

Two commands generalize repetition across the entire grammar rather than being tied to any single operator or motion. The period (`.`) repeats the last change exactly, whatever operator, count, and motion or text object composed it; a user who deletes a word with `daw` and then presses `.` with the cursor on a different word deletes that word too, without retyping the command. This works because Vim records the last change not as a fixed string of keystrokes but as the structured command that produced it, and replays that structure against the new cursor position.

The semicolon (`;`) and comma (`,`) repeat the most recent character-search motion (`f`, `F`, `t`, or `T`, covered in Chapter 5), in the same direction and the reverse direction respectively. Both of these repetition mechanisms exist for the same underlying reason: once a command has been composed once, from the grammar's small set of parts, replaying it should not require recomposing it from scratch, and the grammar accordingly treats "the last thing that happened" as itself a reusable object.

4.7 Composition and the Arithmetic of Coverage

The claim made informally in Chapter 1, that a small number of primitives multiply into a large number of usable commands, can now be stated with more precision. Suppose a reader has learned some number of operators, and some number of motions and text objects that those operators can target. The number of distinct, meaningful commands available is not the sum of these two counts but, to a first approximation, their product: every operator paired with every applicable motion or text object is a distinct, valid command, whether or not that particular pairing has ever been explicitly demonstrated to the reader before.

This is why learning a single new operator (say, the case-conversion operator `gU`, covered in the next chapter) does not add "one more thing to remember" in proportion to how narrow it feels in isolation; it adds one new command for every motion and text object already known, all at once, because `gUw`, `gUiw`, `gUap`, and dozens of other combinations become simultaneously valid and immediately guessable the moment the operator itself is understood. The inverse is equally true: learning a single new text object, once a handful of operators are already known, is not "one more region to remember" but one new target multiplying against every operator that can already be applied to it. This multiplicative structure, more than any specific command covered later, is the practical payoff of having read this chapter, and it is worth returning to explicitly whenever a later chapter introduces something that might otherwise look like an isolated new fact rather than a new multiplier on everything already known.

Chapter 5

Coordinate Systems

A common complaint from newcomers to Vim, encountering its movement commands for the first time, is that there seem to be too many ways to move the cursor, with no evident organizing principle beyond historical accretion. This chapter argues the opposite: Vim's motions are not an unstructured pile of alternatives but several distinct coordinate systems layered over the same buffer, each addressing text at a different granularity, and each internally consistent once recognized as its own system rather than judged against the others. A reader who tries to memorize forty motions as forty independent facts is working much harder than necessary; a reader who recognizes eight or nine small coordinate systems, each with two or three members, is memorizing almost nothing at all.

5.1 Character Coordinates

The finest-grained coordinate system addresses individual characters. `h` and `l` move left and right one character at a time; `0` moves to the first column of the current line and `$` to the last; a count prefixed to either `h` or `l` moves that many characters. This is the coordinate system a reader already met in Chapter 3, and it is deliberately the least distinctive: character-wise motion is the fallback every other coordinate system exists to make less necessary, since moving six words by pressing `l` thirty times is possible but is exactly the kind of low-level, high-repetition operation the word, sentence, and paragraph coordinate systems below are designed to replace with a single keystroke.

5.2 Line Coordinates

A second, closely related system addresses whole lines rather than characters within them. `j` and `k` move down and up one line; `G` moves to the last line of the file, and a count prefixed to `G` (`25G`) moves to that specific line number; `gg` moves to the first line. Line coordinates and character coordinates interact in a way worth noting explicitly: `j` and `k` preserve, where possible, the cursor's original column as it moves between lines of differing length, which is why moving down through a ragged block of text does not steadily drift the cursor toward column zero the way a naive implementation might.

5.3 Word Coordinates

Word motions address a coarser unit than characters, and Vim in fact defines two related but distinct notions of "word." A lowercase `w` moves to the beginning of the next word, where a word is bounded by whitespace or by a transition between alphanumeric and punctuation characters, meaning `don't` is treated as three words (`don`, `'`, `t`) under this definition. An uppercase `W` moves to the beginning of the next WORD (Vim's own capitalized term, adopted here for consistency with the documentation), where a WORD is bounded only by whitespace, treating `don't` as a single unit. `b` and `B` are the backward equivalents of `w` and `W`; `e` and `E` move to the end of the current or next word or WORD rather than its beginning; `ge` and `gE` move backward to the end of the previous word or WORD.

The word/WORD distinction is not a redundancy to be memorized as two arbitrary commands; it is a single coordinate system offered at two different resolutions, chosen depending on whether punctuation should count as a boundary. A programmer navigating through `some_function(arg1, arg2)` will usually want the punctuation-sensitive lowercase forms, since each argument and each punctuation mark is a meaningful unit on its own; a writer navigating through ordinary prose will often find the coarser WORD forms move more efficiently through contractions and hyphenated words without stopping at every internal punctuation mark.

5.4 Sentence and Paragraph Coordinates

Moving outward in granularity again, `(` and `)` move to the beginning of the current or next sentence, and `{` and `}` move to the beginning of the current or next paragraph. Vim's definitions of these units are syntactic rather than semantic: a sentence, for the purposes of these motions, ends at a `.`, `!`, or `?` followed by whitespace or the end of a line (with some further trailing-punctuation handling covered in the built-in help), and a paragraph is delimited by blank lines, not by any deeper notion of topical coherence. This is worth stating plainly because it is a common source of mild disappointment: these motions cannot know that a sentence is "really" still going after an abbreviation like "Dr." followed by a capital letter, and they will treat that period as a sentence boundary regardless. The motions are fast and reliable for what they actually compute, which is punctuation and blank-line structure, and are not attempting anything more linguistically sophisticated than that.

5.5 Section Coordinates

A coarser and considerably less commonly used coordinate system addresses sections, delimited by a form-feed character or, by convention in some file types, by a line beginning with an open brace in the first column. `[[` moves backward to the beginning of the current or previous section and `]]` moves forward to the next. This system predates most modern uses of Vim and is of limited relevance to prose or typical source code, but it persists as a documented motion, and readers working with older-style C source files or documents that use form feeds as chapter separators will find it still functions exactly as originally designed.

5.6 Search Coordinates

Search introduces a coordinate system defined not by any fixed structural unit but by wherever a pattern next matches. `/pattern` searches forward, `?pattern` searches backward, and `n` and `N` repeat the most recent search in the same and opposite direction respectively. A closely related, finer-grained variant restricts the search to the current line and to a single character: `f`*char* and `F`*char* search forward and backward for the next occurrence of *char* on the current line, landing on it, while `t`*char* and `T`*char* do the same but land one character before it (Chapter 4 already introduced `;` and `,` as the repetition commands for this specific family). What unifies search into a single coordinate system, despite these two granularities, is that both forms address text by content rather than by a fixed unit of structure: the destination is wherever the pattern or character is found, not a position defined independently of the buffer's actual contents.

5.7 Screen Coordinates

A further coordinate system addresses position relative to the currently visible screen rather than to the buffer's absolute content. `H` moves to the first line currently visible on screen, `M` to the middle visible line, and `L` to the last visible line; each accepts a count that offsets from the relevant edge rather than jumping to an absolute buffer position. This system exists because "the top of what I am currently looking at" and "line 1 of the file" are frequently different positions, especially in a long file scrolled well past its beginning, and screen coordinates let the user address the former directly without first determining what absolute line number it happens to correspond to.

5.8 Absolute Coordinates

Distinct again from screen-relative position is absolute position within the buffer, addressed by line number directly. `n`G moves to line *n*; `n|` moves to column *n* of the current line; `G` alone (equivalent to specifying the last line) moves to the end of the file, and `gg` (equivalent to line 1) moves to its beginning. This is the coordinate system most directly useful when working from external information, such as a compiler error message or a search tool's output, that reports a specific line number the user needs to reach directly rather than by relative navigation.

5.9 Marks as a Coordinate System

Finally, marks, already introduced in Chapter 2 as a named entity, function grammatically as their own coordinate system once set. `'a` mos to the line of mark `a`; `` `a `` (using the backtick rather than the apostrophe) moves to the exact character position of mark `a` rather than merely its line, a distinction that matters whenever a motion or operator needs to respect the mark's precise column, not just the line it sits on. Two automatically maintained marks are worth knowing from the outset: ``` `` ``` (or, equivalently, `''`) returns to the position

before the most recent jump, and `'.` returns to the position of the most recent change, both covered further in Part IV's chapter on Marks and the jump list. What makes marks a coordinate system rather than simply a storage mechanism is precisely this: once set, a mark is addressable by every operator exactly the way a word or a line number is, filling the same grammatical slot established in Chapter 4.

5.10 Choosing a Coordinate System

The practical skill this chapter is building toward is not memorizing every motion in isolation but developing the reflex to ask, given an editing goal, which coordinate system most directly addresses it. Deleting to the end of a function call is a job for word or bracket-matching coordinates (the latter covered alongside structural regions in Chapter 6); deleting to a specific recalled position elsewhere in the file is a job for a mark; deleting to the next occurrence of a particular character on the current line is a job for `f` or `t`. Fluency in Vim looks, from the outside, like choosing the right motion instantly and without visible deliberation; from the inside, it is closer to routing an editing goal to the coordinate system that already matches its shape, a skill this chapter's organization is intended to make explicit rather than leaving it to be inferred gradually from years of accumulated habit.

Chapter 6

Structural Regions

Chapter 2 introduced the text object as a distinct kind of entity from a motion: where a motion describes a path the cursor could travel, a text object describes a region directly, defined by the structure it belongs to rather than by any particular route to its edges. Chapter 4 placed text objects in the grammar, showing that they fill the same slot a motion fills when paired with an operator. This chapter returns to text objects on their own terms, cataloging the structural regions Vim recognizes and examining the inner/around distinction that runs through nearly all of them.

6.1 The Inner/Around Distinction

Almost every text object Vim defines comes in two forms, distinguished by a prefix letter: `i` for "inner," selecting a structure's content without its delimiters, and `a` for "around," selecting the structure together with its delimiters and, in most cases, one adjacent span of trailing whitespace. `ci(` changes the contents of a pair of parentheses without disturbing the parentheses themselves; `ca(` changes the parentheses and their contents together, removing them entirely if nothing is typed to replace them. This single distinction, learned once, applies almost without exception across every structural region below, which is precisely the kind of multiplicative economy Section 4.7 in the previous chapter described: understanding `i/a` as a general rule, rather than as forty separate memorized command pairs, is what makes the whole catalog tractable.

6.2 Word Regions

`iw` selects the inner word under or after the cursor: the run of alphanumeric or punctuation characters the cursor sits within, using the same word-boundary rules as the `w` motion from Chapter 5. `aw` selects the word together with one adjacent span of whitespace, generally the trailing whitespace if any exists, or the leading whitespace otherwise. `iW` and `aW` apply the same inner/around distinction to Vim's coarser, punctuation-insensitive WORD unit. The practical effect of the whitespace-inclusion rule in the around forms is that repeatedly deleting a word with `daw` does not slowly accumulate stray double spaces the way repeatedly deleting with `diw` would; the around form is generally the correct default for actually removing a word from a sentence, while the inner form is generally correct when the word is about to be replaced with something of a different length via `ciw`.

6.3 Quote Regions

`i"`, `i'`, and `` i` `` select the contents of the nearest double-quoted, single-quoted, or backtick-quoted string on the current line, without the quote marks; `a"`, `a'`, and `` a` `` include the quote marks and, where present, one trailing space. Quote text objects are line-bound in classic Vim: they search only within the current line for the nearest enclosing or following quoted pair, which is a deliberate limitation rather than an oversight, since quote characters are common enough that searching an entire buffer for a matching pair could easily select a much larger, unintended span. A cursor positioned before the first quote character on a line will target the next quoted string that appears later on that same line, which is worth knowing since it means these text objects do not strictly require the cursor to already be inside the quotes to function.

6.4 Bracket and Parenthesis Regions

`i(` and `i)` (equivalently, `ib`, provided as a shorter mnemonic) select the contents of the nearest enclosing pair of parentheses; `i{`, `i}`, and the mnemonic `iB` do the same for curly braces; `i[` and `i]` for square brackets; `i<` and `i>` for angle brackets. The corresponding `a` forms include the delimiters themselves. Unlike quote regions, bracket regions are not restricted to the current line and correctly handle nesting: invoked with the cursor inside a nested pair of parentheses, `i(` selects the innermost enclosing pair, and pressing the same command again while that selection is active (in Visual mode, introduced in Chapter 2 and taken up further throughout Part III) expands the selection outward to the next enclosing pair, a behavior that makes working with deeply nested expressions, common in Lisp-family languages and in deeply nested JSON or configuration data, considerably more tractable than manually counting matched delimiters by eye.

6.5 Tag Regions

`it` selects the content between an opening and closing markup tag, such as the text between `<p>` and `</p>` in HTML or between an opening and closing element in XML, without the tags themselves; `at` includes the tags. This text object is markup-aware in a way the bracket text objects are not: it parses tag names and matches an opening tag to its corresponding closing tag by name, correctly skipping over unrelated intervening tags at different nesting depths, rather than relying on any single, universal delimiter character the way parenthesis matching does. It is consequently one of the more computationally involved text objects Vim provides, and one of the more immediately useful ones for anyone editing HTML, XML, or similar markup on a regular basis, covered further in the context of writing workflows in Chapter 20.

6.6 Sentence and Paragraph Regions

`is` and `as` select the current sentence, using the same sentence-boundary rules as the `(` and `)` motions from Chapter 5, with the around form including trailing whitespace; `ip` and

`ap` do the same for the current paragraph, using the same blank-line-delimited definition the `{` and `}` motions use. These text objects are the direct region-based counterparts of the sentence and paragraph motions already covered, and the relationship between a coordinate system in Chapter 5 and the corresponding structural region here is not a coincidence: wherever a motion moves to the boundary of some structural unit, that unit is generally also available as a text object addressing its interior directly, which is a pattern worth watching for throughout the rest of the book rather than treating each pairing as a separate fact to memorize.

6.7 Block Regions

Visual Block mode, taken up further in Part III, is itself a way of selecting a rectangular region of a buffer directly, by column range across multiple lines, rather than a text object in the operator-plus-target sense used elsewhere in this chapter. It is mentioned here because it fills a structurally similar role: a way of designating a region other than a simple linear span of characters. Where the text objects above select spans defined by delimiters, word boundaries, or blank lines, block selection selects a span defined by column position, useful for editing aligned columns of data or adding the same prefix to several consecutive lines at once.

6.8 User-Defined Regions

Everything covered so far in this chapter is built into Vim, but the inner/around pattern is not closed to further extension. Vimscript (Part VI) permits defining entirely new text objects through custom operator-pending mappings, letting a user or a plugin author extend the same `i`/`a` grammar to structures Vim has no built-in notion of, such as an entire function definition in a specific programming language, a Markdown heading section, or an indentation level. This book, consistent with the plugin-free philosophy stated in the Preface, does not cover any specific third-party text-object definitions, but it is worth knowing the mechanism exists, because it is direct evidence for a claim made repeatedly throughout this chapter: the inner/around distinction is not a fixed, closed list of special cases Vim happens to support, but a general grammatical pattern, available to be extended to any structural unit a user can define programmatically, using exactly the same two-letter vocabulary already learned for the built-in cases above.

6.9 Structural Regions and the Grammar

It is worth closing this chapter, and with it Part II, by returning explicitly to where structural regions sit within the larger rule established in Chapter 4. A text object is never used alone in ordinary editing; it is always the target half of an operator-target pair, or the boundary of a Visual-mode selection about to be acted on. Everything catalogued in this chapter multiplies against everything catalogued in the next chapter's treatment of operators, exactly as Section 4.7 of Chapter 4 predicted it would: roughly a dozen structural regions, each with

an inner and around form, multiplied against a handful of core operators, already yields several dozen distinct, meaningful commands, the overwhelming majority of which this book will never explicitly demonstrate, because by this point in the book they no longer need to be. That is the payoff Part II was written to deliver.

Part III

Transformations

Chapter 7

Operators

Chapter 4 introduced the operator as a grammatical category: a command that waits for a motion or text object before it does anything. This chapter examines the operators themselves, the actual transformations Vim performs once that target is supplied. Where Part II was concerned with the shape of the sentence, this chapter and the two that follow it are concerned with what the verb can mean.

7.1 Deletion

`d` removes the text spanned by its target and places it in a register (the unnamed register by default, or a named one if specified, per Chapter 10). `dw` deletes to the beginning of the next word; `d$` (or its shorthand `D`) deletes to the end of the line; `dd` deletes the current line as a unit, per the doubling convention established in Chapter 4. Deletion in Vim is never destructive in the sense of discarding the removed text outright: because the deleted span is always written to a register first, undoing a deletion is rarely even necessary, since the same text can usually be retrieved with a put command instead, a point developed further once registers themselves are covered in Part IV.

7.2 Change

`c` behaves exactly like `d` with one addition: after removing the spanned text, it leaves the editor in Insert mode at the point of removal, ready for replacement text to be typed. `cw` changes to the end of the current word; `ciw` changes the inner word text object regardless of cursor position within it; `cc` changes the entire current line, deleting its content but preserving its indentation, which distinguishes `cc` from a plain `dd` followed by `O` in a way that is easy to overlook until it is needed.

The relationship between `c` and `d` is worth stating explicitly, because it is a small but genuine exception to the otherwise uniform operator grammar: `c` is not simply "delete, then separately enter Insert mode," which would be two commands composed by the user, but a single operator whose defined behavior includes that mode transition as part of what the operator itself does. This matters practically because it means `c` cannot be meaningfully separated into a delete-operator-plus-insert-command pair for the purposes of the period-repeat command from Chapter 4: pressing `.` after `ciw` followed by typed replacement text repeats the entire compound action, deletion and the subsequent insertion together, against a new target.

7.3 Yanking

`y` copies the spanned text into a register without removing it from the buffer, the direct analogue of `d` for the reader who wants to duplicate text rather than relocate it. `yw` yanks to the beginning of the next word; `yy` yanks the current line. Because deletion already writes to a register as a side effect, a common point of confusion for newcomers is not understanding why an explicit yank command is needed at all; the answer is that deletion's register write is incidental to removing the text, while yank's entire purpose is that write, with no accompanying removal, and the two operators exist separately because "copy without altering the buffer" and "remove, incidentally retaining a copy" are genuinely different operations that happen to share an underlying storage mechanism.

7.4 Case Conversion

`gU` uppercases the spanned text and `gu` lowercases it; `g~` toggles the case of each character in the span independently. `gUiw` uppercases the inner word under the cursor; `guaw` lowercases a word and its surrounding whitespace region (the whitespace itself being unaffected by a case operation, since it contains no letters, but included in the span for consistency with how `aw` is defined). These operators are a clean illustration of Section 4.7's multiplicative claim from Chapter 4: nothing about case conversion required a special case-conversion grammar distinct from deletion or yanking; it required only a new operator, immediately usable against every motion and text object already known.

7.5 Indentation

`>` increases indentation by one shift width (configured with the `shiftwidth` option introduced in Chapter 3) across the spanned lines, and `<` decreases it. Because indentation is inherently line-oriented, these two operators are conventionally doubled (`>>`, `<<`) to affect the current line, and prefixed with a count to affect several lines at once (`3>>` shifts the next three lines), rather than commonly paired with the finer-grained motions used for deletion or yanking; shifting half a line by one indent level is rarely a meaningful operation, so the indentation operators are, in practice, used almost exclusively in their doubled or Visual-mode forms.

`=` is a related but distinct operator: rather than shifting by a fixed number of shift widths, it invokes Vim's indentation logic (either its built-in, filetype-aware rules or an external program specified by the `equalprg` option) to compute what the correct indentation of the spanned lines should be, and applies that instead. `=i{` auto-indents the contents of the current code block according to the file's own indentation rules, a genuinely different operation from `>i{`, which would shift the block by one fixed amount regardless of what the surrounding code's indentation logic considers correct.

7.6 Joining

`J` joins the next line onto the end of the current one, inserting a single space at the join point (unless the current line already ends in whitespace, or the next line begins with a closing parenthesis, in which case Vim's built-in joining logic omits the extra space to avoid producing an obviously wrong result). `gJ` performs the same join without inserting any space, a literal concatenation. A count prefixed to either form joins that many lines at once: `3J` joins the current line with the following two. Unlike most of the operators in this chapter, `J` does not take a motion or text object argument; it operates on a fixed unit (the following line, or, with a count, the following several lines) rather than on an arbitrary span, which places it slightly outside the general operator-target grammar even though it shares the operator category by convention.

7.7 Filtering Through the Shell

`!` is an operator in the formal sense used throughout this chapter, but one whose target is not merely a motion or text object; it sends the spanned text out of Vim entirely, to an external shell command, and replaces it with that command's output. `!}sort` takes the current paragraph, pipes it through the `sort` command, and replaces the paragraph with the sorted result. This operator is the earliest concrete point of contact in this book between the operator grammar covered so far and the wider Unix integration that Part VII treats in depth; it is included in this chapter, rather than deferred entirely, because it obeys the same operator-target grammar as every other transformation here, differing only in where the actual transformation is computed.

7.8 Formatting

`gq` reformats the spanned lines to wrap at the width specified by the `textwidth` option, and `gw` does the same while preserving the cursor's position rather than moving it to the end of the formatted text. `gqip` reformats the current paragraph, a genuinely common operation when editing prose (covered further in Chapter 20's treatment of writing workflows) where a line has grown too long or a paragraph has been edited into ragged, inconsistent line lengths that no longer wrap cleanly.

7.9 Operators as a Closed but Extensible Class

The operators catalogued in this chapter, deletion, change, yank, case conversion, indentation, joining, shell filtering, and formatting, are not an arbitrary list; they are close to the complete set of built-in, general-purpose transformations Vim applies to a span of text identified by the grammar of Part II. New operators can be defined through Vimscript (Part VI), in exactly the same way new text objects can be, but this book's plugin-free scope means the operators above represent, practically speaking, the full built-in vocabulary a reader needs to combine with every motion and text object already covered. What Chapter 4 promised

as an economical, multiplicative grammar should, by the end of this chapter, feel concretely realized: roughly eight operators, each multiplying against dozens of motions and text objects, already account for several hundred distinct, meaningful commands, the overwhelming majority of which have not been individually demonstrated anywhere in this book and do not need to be.

Chapter 8

Inserting and Replacing

Every operator covered in Chapter 7 removes, copies, or transforms text that already exists in the buffer. This chapter turns to the opposite case: adding new text, and the family of Normal-mode commands that transition into Insert mode (or one of its variants) at a specific, deliberately chosen point, each differing only in where that point is and what, if anything, is removed before typing begins.

8.1 Insert and Append

`i` enters Insert mode with the cursor unchanged, so that typed text appears before whatever character the cursor was on. `a` (append) enters Insert mode one character to the right, so that typed text appears after the character the cursor was on, which is the natural choice when adding text immediately following the current cursor position rather than immediately preceding it, and is also the only sensible option when the cursor is on the last character of a line and the intent is to continue typing past it. `I` and `A` are the line-relative extremes of the same pair: `I` inserts at the first non-blank character of the current line regardless of where the cursor happened to be, and `A` appends at the very end of the line. A reader who has internalized the coordinate systems of Chapter 5 may notice that `I` and `A` are, in effect, `^i` and `$a` respectively, expressed as dedicated single-key commands because moving to a line's edge before inserting text is common enough to deserve its own shorthand.

8.2 Opening Lines

`o` opens a new, empty line immediately below the current one and enters Insert mode on it; `O` does the same immediately above. Both commands respect the current line's indentation when the `autoindent` or filetype-specific indentation settings are active, carrying that indentation forward onto the new line automatically rather than leaving the reader to type it manually, which is a small detail but one that matters considerably when writing indentation-sensitive code.

8.3 Replace Mode

`R` enters Replace mode, a mode distinct from ordinary Insert mode in that each typed character overtypes the character currently under the cursor rather than pushing it rightward, continuing until `escape` returns to Normal mode. This is the direct keyboard analogue

of the "overtype" or "insert versus overwrite" toggle familiar from many non-modal text editors, except that in Vim it is a genuine mode with its own identity rather than a global toggle affecting how Insert mode itself behaves. A single-character variant, `r`*char*, replaces only the character under the cursor with *char* and returns immediately to Normal mode without ever entering Replace mode at all, which makes it the more commonly used of the two for small, one-character corrections.

8.3.1 Virtual Replace

`gR` enters Virtual Replace mode, a variant of Replace mode aware of virtual columns, meaning it correctly overtypes through tab characters and multi-width characters by their visual column position on screen rather than their raw byte or character position in the buffer. This distinction rarely matters when working with plain, tab-free text, but becomes important when replacing text in a file that mixes tabs and spaces for indentation, where ordinary Replace mode's character-position overtyping can produce visually misaligned results that Virtual Replace mode avoids.

8.4 Substituting

`s` deletes the character under the cursor and enters Insert mode at that point, equivalent to `cl` (change one character) but shorter to type; `S` deletes the entire current line's content, preserving indentation, and enters Insert mode, equivalent to `cc` from the previous chapter. These are not new grammatical constructs so much as convenient, frequently used shorthand for operator-motion combinations already covered, included here because they are common enough in practice to be worth naming directly rather than leaving the reader to rediscover them as a special case of `c`.

8.5 Completion

While in Insert mode, `Ctrl-n` and `Ctrl-p` trigger keyword completion, searching the current buffer (and, depending on configuration, other open buffers, included files, and tag files) for words beginning with whatever has been typed so far, and presenting matches in a pop-up menu navigable with the same two keys. This is Vim's built-in completion mechanism, present without any plugin, and while considerably less sophisticated than the language-aware completion offered by modern editors with dedicated language servers, it is frequently sufficient for completing a variable name, a function name, or any other identifier that has already appeared elsewhere in the buffer, and it is worth knowing as a baseline capability before reaching for any external tooling.

8.6 Digraphs and Special Characters

`Ctrl-k` followed by a two-character digraph code inserts the corresponding special character: `Ctrl-k e'` inserts é, for instance, using a mnemonic two-letter code defined in Vim's digraph table (viewable with `:digraphs`) rather than requiring the reader to know or type

a raw Unicode code point. For characters without a convenient digraph, `Ctrl-v` followed by a decimal, octal, hexadecimal, or Unicode code point inserts that exact character directly: `Ctrl-v u 00e9` inserts the same é by its Unicode code point. `Ctrl-v` additionally serves a second, more mundane purpose while inserting text: pressing it before any character that would otherwise carry a special meaning (a literal tab when `expandtab` would normally convert it to spaces, for instance) inserts that character's literal form instead, bypassing whatever special handling would otherwise apply.

8.7 Insert Mode Is Still Insert Mode

It is worth closing this chapter by noting what has not changed across every command covered here: `i`, `a`, `I`, `A`, `o`, `O`, and the insert-producing forms of `c` and `s` all lead to the same Insert mode, differing only in the cursor position and the state of the buffer at the moment that mode is entered. None of them constitute a separate typing mode with its own rules; once inside Insert mode, ordinary typing behaves identically regardless of which command was used to arrive there, and `escape` returns to Normal mode the same way from all of them. What this chapter has actually catalogued, in other words, is not eight different ways of typing text, but eight different ways of choosing where typing should begin, which is a considerably smaller thing to have learned than the length of this chapter might suggest, and is itself a small instance of the same economy of primitives this book has been arguing for since Chapter 1.

Chapter 9

Undo and Time

Every transformation covered in Chapters 7 and 8 is reversible, and this chapter is concerned with that reversal: not as an emergency escape hatch, though it functions as one, but as a navigable structure in its own right, one with more depth to it than the simple linear "step backward" model a reader arriving from other editors might expect.

9.1 Basic Undo and Redo

`u` undoes the most recent change, and can be pressed repeatedly to step back through successive earlier changes, one at a time. `Ctrl-r` redoes a change that was just undone, stepping forward again. `U` is a distinct, older command that undoes all changes made to the current line since it was last visited, which behaves unlike a simple repeated `u` in one specific respect worth flagging: pressing `U` a second time undoes the effect of the first `U`, restoring the line to the state `U` had just undone it from, rather than continuing to undo further back, which makes `U` function as a toggle on a single line's state rather than as a general step-backward command.

What counts as a single "change" for the purposes of `u` is worth stating explicitly, since it is not always the same granularity a reader might assume. An entire Insert-mode session, from the moment Insert mode is entered until `escape` returns to Normal mode, counts as one change, however much text was typed during it; a single `u` after a long paragraph has been typed undoes the entire paragraph at once, not the most recently typed character. A single operator-motion command, such as `daw`, also counts as one change. This granularity is generally the intuitive one in practice, since it corresponds to what a user would describe as "one edit" in ordinary language, but it is worth knowing explicitly for the cases where it matters, such as a very long Insert-mode session that the user may want to partially, rather than entirely, undo.

9.2 The Undo Tree

The model implied by "step backward, step forward" is linear, and it is the model most editors implement: a single sequence of states, with undo and redo moving a pointer backward and forward along it. Vim's undo history is not linear; it is a tree. If the user undoes several changes and then makes a new edit rather than redoing, the changes that were undone are not discarded, as they would be in a strictly linear model. They remain in the undo history

as a separate branch, reachable again later, while the new edit begins a second branch from the same point.

This matters concretely in a scenario every long editing session eventually produces: a paragraph is deleted, several unrelated edits are made elsewhere in the file, and only then does the user realize the original deletion should not have happened. In a linear undo model, undoing back to before the deletion would also undo all the unrelated edits made afterward, forcing a choice between recovering the paragraph and keeping the later work. Vim's tree-structured undo history does not force this choice, because the later edits and the deleted paragraph live on different branches of the same tree, both still present, both still reachable.

`g-` and `g+` move backward and forward through the undo history by *time* rather than by tree position, meaning they follow the sequence in which changes actually occurred, across branches, rather than following the parent-child structure `u` and `Ctrl-r` follow within a single branch. `:undolist` displays the tree's branch points directly, each labeled with a number, and `:undo n` jumps directly to the state after change number *n*, which is the most direct way to navigate to a specific branch once `:undolist` has been consulted to identify it.

9.3 Persistent Undo

By default, a buffer's undo history exists only for the duration of the editing session; closing the file and reopening it later begins with no undo history at all, even if the file was previously edited and saved. `set undofile`, placed in the vimrc introduced in Chapter 3, changes this: Vim writes the undo history to a separate file on disk (by default alongside the edited file, or in a centralized directory if `undodir` is set), and reloads that history automatically the next time the same file is opened, meaning `u` can undo changes made in a previous, entirely separate editing session, potentially days or weeks earlier, exactly as though that session had never ended. This is one of the more consequential single-line additions to a minimal vimrc, and worth adding immediately once it is known to exist, since its absence is only ever noticed at the moment it would have been useful and no longer can be.

9.4 Swap Files and Crash Recovery

Separately from the undo history, Vim maintains a swap file for each buffer currently being edited (typically named `.filename.swp` in the same directory as the file, unless `directory` is configured otherwise), recording changes as they are made so that an abnormal termination, a crash, a lost SSH connection, a killed process, does not lose the in-progress edits entirely. On a normal, clean exit, the swap file is deleted automatically; its continued presence when a file is reopened is itself the signal that a previous session did not end cleanly, and Vim will warn the user of this directly rather than silently ignoring it.

When that warning appears, `vim -r` *filename* recovers the buffer's state from the swap file as it stood at the last recorded point before the abnormal termination, rather than opening the file fresh from its last saved state on disk, which may be considerably further be-

hind. Recovery should generally be followed by writing the recovered buffer to disk under a new name first, inspecting it, and only then deciding whether to overwrite the original file, since a swap file's contents, while usually reliable, are not guaranteed to reflect every last keystroke before the crash occurred.

9.5 Undo as Confidence

The material in this chapter is not, by itself, part of the operator-motion grammar Part II established, and it does not multiply against operators or text objects the way the material in Chapters 7 and 8 does. It is included in Part III regardless, immediately after the transformations it exists to reverse, because its practical effect on how the rest of this book should be read is significant: every operator, every insertion, every substitution covered up to this point is safe to attempt experimentally, on real files, because `u` is always available, the undo tree never discards a state the user has not explicitly abandoned, and persistent undo, once enabled, extends that safety net across sessions rather than confining it to the current one. A reader who has internalized this chapter should feel, for the remainder of the book, considerably less hesitant about trying an unfamiliar command on a real file than they otherwise might.

Part IV

State

Chapter 10

Registers

Part IV turns from transformation to memory: not the editor's undo history, already covered in Chapter 9, but the several distinct mechanisms Vim provides for holding something so it does not have to be reconstructed from scratch a moment later. Three such mechanisms recur throughout the rest of this book, and it is worth naming the pattern once, here, before treating each separately: a register externalizes a fragment of *text*; a mark, taken up in Chapter 11, externalizes a *position*; and a macro, taken up in Part VI, externalizes a *trajectory of actions*. Each solves the same underlying problem, that working memory is limited and unreliable, at a different grain. This chapter takes the finest grain of the three.

10.1 The Unnamed Register

Every deletion, change, and yank writes its text somewhere, and in the absence of an explicit register name, that somewhere is the unnamed register, addressed as `"` and read by an unadorned `p` or `P`. This is the register a new Vim user interacts with constantly without necessarily knowing its name: `dw` followed by `p` moves a word, because the deletion wrote to the unnamed register and the put read from it, with no register name ever typed by the user. The unnamed register is overwritten by nearly every subsequent deleting or yanking operation, which is the single most common source of a specific, frustrating mistake: yanking a piece of text, performing an unrelated deletion in order to navigate somewhere else, and then discovering that the deletion has silently overwritten the text that was meant to be pasted. Named registers, covered next, exist largely to prevent exactly this.

10.2 Named Registers

Twenty-six named registers, addressed `"a` through `"z`, hold text independently of the unnamed register and independently of each other. `"ayy` yanks the current line into register `a`; `"ap` puts its contents after the cursor. Because these registers are not touched by ordinary deletions and yanks that do not explicitly name them, they provide exactly the durability the unnamed register lacks: text can be stored in register `a`, several unrelated edits can be performed, and register `a`'s contents will still be there when needed.

A register name given in uppercase, rather than lowercase, does not address a different register; it appends to the same one rather than overwriting it. `"Ayy` appends the current line to whatever register `a` already contains, rather than replacing it. This is the mechanism behind a common and genuinely useful workflow: collecting several non-adjacent lines

from different parts of a buffer into a single register, one appending yank at a time, before putting them all at once somewhere else.

10.3 Numbered Registers

Registers `"1` through `"9` are maintained automatically by Vim rather than addressed deliberately by the user in the way named registers are. Register `"1` holds the most recent deletion of a line or larger span; each subsequent such deletion shifts the previous contents of `"1` into `"2`, the previous contents of `"2` into `"3`, and so on, up to `"9`, at which point the oldest entry is discarded. This gives the numbered registers the character of a small, automatic undo history specifically for deleted text, distinct from the undo tree of Chapter 9: `"3p` retrieves a deletion from three deletions ago, without needing to have named a register at the time of that deletion. Small, sub-line deletions (a single character, or a span smaller than a full line) are handled differently, covered next.

10.4 The Small Delete Register

A deletion smaller than one full line, such as `x` or `dw` confined to a single line, is written to the small delete register, addressed `"-`, rather than to the numbered registers described above. This separation exists so that a long sequence of small, incidental deletions (correcting typos, removing individual characters) does not flood the numbered registers and push out a more significant, larger deletion the user may still want to retrieve. The practical consequence is that `"1p` is reliable for retrieving a recent line-or-larger deletion specifically, uncontaminated by whatever small corrections happened to be made afterward.

10.5 Read-Only Registers

Several registers are maintained automatically by Vim and cannot be written to directly; they can only be read. `":` holds the most recently executed Ex command, useful for retrieving and re-executing or adapting a command typed a moment ago without having to retype it from scratch. `"/` holds the most recent search pattern, which is what makes it possible to insert the current search term directly into a substitution command, covered in Chapter 14, without retyping the pattern. `".` holds the most recently inserted text, distinct from the repeat command covered in Chapter 4 in that it holds the literal text itself, retrievable and insertable elsewhere, rather than replaying the insertion as an action. `"%` holds the name of the current file, and `"#` the name of the alternate file (the buffer's previous file, formalized further in Chapter 12). These registers are worth knowing exist even if a reader does not use them daily, because each answers a specific, recurring "how do I get Vim to remember this without retyping it" question the moment that question arises.

10.6 The Expression Register

The expression register, addressed `"=`, is not a storage location in the same sense as the registers above; reading from it evaluates a Vimscript expression, typed at a prompt, and returns the result as text. In Insert mode, `Ctrl-r =` opens this prompt directly, and typing an expression such as `5*12` and pressing return inserts the literal text `60` at the cursor. This is a small preview of the Vimscript material in Part VI, included here because it is, formally, a register like any other: something `"=p` or `Ctrl-r =` can read from, even though what it returns is computed rather than stored.

10.7 Clipboard Registers

On a Vim build compiled with clipboard support (confirmed, per Chapter 3, with `vim --version`), two further registers bridge Vim's internal register system to the operating system's own clipboard. `"*` corresponds to the primary selection on X11 systems (the text most recently selected with the mouse, pasted with a middle-click) or to the standard clipboard on Windows and macOS; `"+` corresponds specifically to the standard copy-paste clipboard on systems that distinguish the two (chiefly X11-based Linux desktops). `"+yy` yanks the current line directly to the system clipboard, from which it can be pasted into any other application; `"+p` pastes the system clipboard's contents into the buffer.

This is also the appropriate place to note a caution already raised in Chapter 3: clipboard register behavior is the single most environment-dependent piece of the register system, varying with the build of Vim, the terminal emulator, and, on Windows, whether Vim is running under Git Bash, WSL, or natively. A reader whose `"+y` does not appear to reach the system clipboard should suspect the environment before suspecting the register syntax itself, and confirm clipboard support with `vim --version` as a first step.

10.8 The Black Hole Register

The black hole register, addressed `"_`, discards whatever is written to it and, symmetrically, returns nothing when read. `"_dd` deletes the current line without overwriting any other register, unnamed or otherwise, which is the practical use case: performing a deletion purely to remove text, with no intention of ever pasting it back, without that deletion's side effect of clobbering whatever the unnamed register was previously holding. A reader who has been burned once by the unnamed-register overwrite problem described at the start of this chapter will likely find `"_d` becoming a reflexive habit for exactly that situation.

10.9 Registers as a System, Not a List

The register catalog in this chapter is long enough to read, at first pass, like an arbitrary collection of special cases: an unnamed default, twenty-six named slots, nine numbered slots with automatic shifting, a small-delete carve-out, several read-only sources, an expression evaluator, two clipboard bridges, and a discard sink. It is worth closing by restating what

unifies them: every one of these is addressed with the same `"`*name* syntax, readable by the same `p/P` commands and writable, where writable at all, by the same operator-plus-register-prefix syntax covered throughout this chapter. A reader who has learned to yank into register `a` has, at zero additional syntactic cost, also learned how to yank into the system clipboard, retrieve a three-deletions-ago line, or discard a deletion cleanly, because all four operations are the same grammatical move applied to a different register name. This is the same multiplicative economy Chapter 4 established for operators and motions, applied here to registers instead.

Chapter 11

Marks

Where a register externalizes a fragment of text, a mark externalizes a position: a specific line and column, given a name, to which the cursor can return directly without scrolling or searching. Marks were introduced as an entity in Chapter 2 and treated as a coordinate system in Chapter 5; this chapter is where they are covered in full, alongside the several kinds of position Vim tracks automatically without the user ever having set a mark by hand.

11.1 Local Marks

A local mark, set with `m`*letter* using a lowercase letter, is specific to the buffer in which it was set. `ma` sets mark `a` at the cursor's current position; `'a` later returns to the first non-blank character of that mark's line, and `` `a `` (the backtick form) returns to its exact character position rather than merely its line. This distinction, already noted in Chapter 5, matters most when a mark is used as a motion target for an operator: `d`a` deletes to the mark's precise column, while `d'a` deletes to the first non-blank character of the mark's line, which can be a meaningfully different span depending on what precedes that column.

Local marks persist for the duration of the editing session and, if `shada` (or, on older Vim versions, `viminfo`) is configured to retain them, across sessions as well, making it possible to set a mark, close Vim entirely, and return to that exact position days later.

11.2 Global Marks

A global mark, set with an uppercase letter, is addressable from any buffer, not merely the one it was set in. `mA` sets global mark `A` at the cursor's position in the current file; from an entirely different buffer, `'A` or `` `A `` jumps directly to that position, opening the original file if it is not already loaded. This is the mechanism for maintaining a small set of named, cross-file bookmarks in a large project: a handful of files or specific lines within them, each given a memorable uppercase letter, reachable from anywhere without first navigating to the right file by hand.

11.3 Automatic Marks

Several marks are maintained by Vim without any explicit `m`*letter* command from the user, each recording a specific kind of recent event.

`` (or, equivalently, '') holds the cursor's position immediately before the most recent jump, where a jump is any of the several motions covered below that move the cursor a nontrivial distance rather than incrementally. Pressing `` a second time returns to wherever the cursor was before the first press, making it function as a toggle between two positions of interest, which is frequently exactly what is needed when comparing two specific points in a file.

'. holds the position of the most recent change to the buffer, useful after a search or a scroll has moved the cursor away from an edit the user wants to return to. '^ holds the position of the cursor the last time Insert mode was exited, which is subtly different from '. in files where the two do not coincide, such as after a change that itself involved further cursor movement before returning to Normal mode. '[and '] hold the start and end of the most recently changed or yanked text, which is what makes it possible to immediately select or otherwise act on exactly the span a previous command just operated on, without having to identify its boundaries by eye.

11.4 The Jump List

Beyond the single automatically maintained `` mark, Vim keeps a longer jump list: a history of positions the cursor has jumped from, navigable with Ctrl-o (older) and Ctrl-i (newer, since Ctrl-i and Tab share a physical key on most keyboards). What counts as a jump for the purposes of this list is a specific, defined set of motions, generally those that move the cursor a nontrivial or unpredictable distance: searches, the G and gg absolute-position motions from Chapter 5, paragraph and section motions, and mark jumps themselves, among others; small, incremental motions like h, j, k, l, or w do not add an entry.

:jumps displays the list directly, with the current position marked. The practical value of the jump list over the single `` mark is depth: where `` toggles between exactly two positions, Ctrl-o can be pressed repeatedly to retrace an entire recent navigation history, which is particularly useful after following a chain of searches or tag jumps several steps deep and wanting to retrace that chain rather than jump directly back to its start.

11.5 The Change List

A related but distinct history, the change list, tracks positions where the buffer's content was actually modified, rather than positions the cursor merely visited. g; moves to the previous position of change, and g, moves to the next, cycling through this list independently of the jump list covered above. Where the jump list answers "where was I recently," the change list answers "where did I recently edit," and the two frequently diverge, particularly in a session that involves substantial searching or reading relative to the amount of actual editing done.

11.6 The Alternate File

Vim maintains a notion of the alternate file: generally, the most recently edited buffer other than the current one. Ctrl-^ (or, equivalently, :e #) switches directly to the alternate file,

toggling back and forth between two files with a single keystroke each time. This is distinct from marks in that it is not a position within a file but a reference to an entire other buffer, and it is distinct from the jump list in that it tracks only a single most-recent alternate rather than a history; it is included in this chapter because it is addressed through the same register mechanism noted in Chapter 10 (the read-only `"#` register holds the alternate file's name) and functions, in practice, as a lightweight complement to the global marks covered earlier for the specific case of two files being edited in close alternation.

11.7 Marks as Durable Coordinates

What unifies every mechanism in this chapter, deliberate marks and automatic ones alike, is that each answers a version of the same question: given that the cursor has moved away from somewhere of interest, how does the user get back without re-deriving the position from scratch. A deliberately set mark answers this for a position the user identified in advance as worth returning to; the jump list and change list answer it retroactively, for positions the user did not think to mark ahead of time but wants to revisit anyway; the alternate file answers a coarser version of the same question at the level of an entire buffer rather than a position within one. Taken together with the register material of Chapter 10, this chapter completes the picture of Vim's memory mechanisms below the level of the undo tree: a position is never truly lost, even when no mark was deliberately set for it, provided the reader knows which of these several histories to consult.

Chapter 12

Buffers, Windows, Tabs, Sessions

Chapter 2 named buffers, windows, and tabs as distinct entities and warned against conflating them: a buffer is loaded content, a window is a viewport onto a buffer, and a tab page is a saved arrangement of windows. This chapter returns to that distinction with the actual commands for managing each, plus sessions, which extend the same idea of "an arrangement worth returning to" across the boundary of a single editing session entirely.

12.1 Managing Buffers

`:ls` (or its synonym `:buffers`) lists every buffer currently loaded, whether or not it is visible in any window, each entry numbered and flagged with its status: `%` for the buffer in the current window, `#` for the alternate buffer from Chapter 11, `a` for active (loaded and visible), `h` for hidden (loaded but not currently displayed anywhere), and `+` for a buffer with unsaved modifications. `:b`*n* switches the current window to buffer number *n*, and `:b`*name* switches to a buffer by a (partially matched) filename, which is generally the faster of the two in practice since it does not require first consulting `:ls` to look up a number. `:bnext` and `:bprevious` (abbreviated `:bn` and `:bp`) cycle through the buffer list in order.

A buffer becomes hidden, rather than closed, when its window is closed while the buffer itself still has unsaved changes Vim is unwilling to discard silently, or when `:hide` or an equivalent command is used deliberately. A hidden buffer retains its full content and undo history; it is simply not currently displayed in any window. `:bd` (buffer delete) removes a buffer from the buffer list entirely, which is a distinct and more final operation than merely closing the window that was displaying it, and will refuse to proceed if the buffer has unsaved changes unless forced with `:bd!`.

12.2 Splitting Windows

`:split` (or `:sp`) divides the current window horizontally, creating a new window above the existing one, both displaying the same buffer initially. `:vsplit` (or `:vsp`) divides vertically instead. Either command can be given a filename, `:split file.txt`, to open a different file in the new window directly rather than duplicating the current buffer into it. `Ctrl-w s` and `Ctrl-w v` are the Normal-mode equivalents, useful for splitting without dropping into Command-line mode at all.

Once split, `Ctrl-w` followed by a direction key (`h`, `j`, `k`, or `l`) moves the cursor to the window in that direction, the same four keys used for character-wise motion in Chapter 5, re-

purposed here for window-wise movement, consistent with the ergonomic argument from Chapter 1 that the home row should serve as many purposes as reasonably possible rather than requiring the hand to travel for a conceptually related but differently scaled operation. `Ctrl-w w` cycles to the next window regardless of direction, and `Ctrl-w c` closes the current window (distinct from `:bd`, since closing a window does not necessarily delete the buffer it was showing, per the hidden-buffer behavior above).

12.3 Resizing and Rearranging

`Ctrl-w =` resizes all windows to be as equal in size as the current layout permits, a useful reset after manual resizing has left the screen unevenly divided. `Ctrl-w +` and `Ctrl-w -` grow and shrink the current window's height by one line (prefixed with a count for a larger adjustment, following the same count convention established in Chapter 4); `Ctrl-w <` and `Ctrl-w >` do the same for width. `Ctrl-w r` rotates the windows in the current layout, and `Ctrl-w x` exchanges the current window with the next one, both useful for correcting a split that was created in a less convenient order than intended without having to close and reopen anything.

12.4 Tabs

A tab page, as established in Chapter 2, holds an entire arrangement of windows, not a single file. `:tabnew` (optionally given a filename) opens a new tab; `gt` and `gT` move to the next and previous tab respectively, and `n gt` moves directly to tab number *n*. `:tabclose` closes the current tab, and `:tabonly` closes every tab except the current one. A count prefixed to `:tabnew`, or the command `:tabnew` issued repeatedly, has no relationship to the count-prefixed motions of Chapter 5; the parallel keystroke vocabulary (`g` followed by a letter, echoing `g;` and `g,` from the change list in Chapter 11) is a naming convention rather than a grammatical extension of the operator-motion rule from Part II.

The practical case for tabs over simply splitting one very large window many times is organizational: a tab groups a set of windows around a single coherent task (one project's relevant files, arranged in whatever split configuration suits that task), and switching tabs switches that entire grouping at once, rather than requiring individual windows to be closed and reopened as the task changes. A reader working on two unrelated pieces of a project simultaneously will generally find two tabs, each internally split as needed, more legible than one tab split many ways across both tasks at once.

12.5 Sessions

Everything covered in this chapter so far, the buffer list, the window layout, the tab arrangement, and each window's cursor position and view, can be captured as a unit and restored later. `:mksession` (optionally given a filename, defaulting to `Session.vim`) writes the current arrangement to a Vimscript file; running Vim with `vim -S` *filename*, or sourcing that file from within a running session with `:source`, restores the arrangement exactly, reopening every buffer, recreating every split and tab, and repositioning every cursor.

Sessions are the natural endpoint of the buffer, window, and tab material in this chapter, because they answer the same "how do I get back to this without reconstructing it by hand" question that motivated marks and registers in the two preceding chapters, applied now to an entire working environment rather than a single position or fragment of text. A reader working across several distinct projects, each with its own characteristic arrangement of files and splits, can maintain a separate session file per project and restore the full working context for any of them with a single command, rather than manually reopening and re-splitting the same handful of files each time work resumes.

12.6 Buffers, Windows, Tabs, and Sessions as a Single Hierarchy

It is worth closing Part IV by restating the hierarchy this chapter has covered, since it is easy to lose track of amid the individual commands: a session contains tabs, a tab contains windows, and a window displays a buffer, which may or may not correspond to a file on disk, per the distinction drawn in Chapter 2. Each level of this hierarchy has its own management commands, largely independent of the levels above and below it, and a reader disoriented by an unfamiliar Vim session (an unexpected number of open windows, an unfamiliar tab arrangement) can generally resolve that disorientation by asking, in order, which buffer is currently active, which window it is displayed in, and which tab that window belongs to, rather than treating the whole arrangement as an undifferentiated puzzle to be solved at once.

Part V

Ex

Chapter 13

Ex as a Language

Every command covered so far in this book has belonged to Normal mode: a keystroke or short sequence of keystrokes, interpreted immediately, acting on the buffer at or near the cursor. Part V turns to the other half of Vim's grammar, previewed as far back as Chapter 2's claim that Normal-mode keystrokes and Ex commands are two surfaces of the same underlying language rather than two unrelated systems coexisting in one editor. This chapter makes that claim concrete, by treating Ex the way Part II treated Normal mode: not as a list of commands to memorize individually, but as a grammar with its own small set of composable parts.

13.1 The Basic Form

An Ex command is typed in Command-line mode, entered from Normal mode with `:`, and executed on pressing return. Its general form is:

```
:[range]command[!] [arguments]
```

A range, covered fully below, restricts the command to a specific span of lines rather than the default of the current line alone; the optional `!` modifies certain commands' behavior, typically toward a more forceful or less cautious variant (`:q!` discards unsaved changes rather than warning about them, as seen already in Chapter 3); and arguments supply whatever further information the specific command requires. `:w` writes the current buffer with no range and no arguments; `:5,10w partial.txt` writes only lines 5 through 10, to a different file than the one currently being edited, using both a range and an argument at once.

This chapter's central claim mirrors Chapter 4's treatment of Normal mode directly: a comparatively small number of ways to specify a range, combined with a comparatively small number of commands that accept one, produces a large number of meaningful combinations, most of which this book will not individually demonstrate because, by the end of this chapter, they will not need to be.

13.2 Addresses

An address identifies a single line. The simplest addresses are absolute line numbers: `:10` alone, entered as an Ex command, moves the cursor to line 10. `.` addresses the current line and `$` the last line of the buffer, mirroring the Normal-mode motions of the same names

from Chapter 5 (this correspondence is not decorative; Ex addresses and Normal-mode motions share vocabulary because they address the same underlying structure from two different surfaces).

A mark, from Chapter 11, is also a valid address: `'a` addresses the line containing mark `a`. A search pattern enclosed in slashes or question marks is likewise a valid address: `/pattern/` addresses the next line matching *pattern* after the current position, searching forward, and `?pattern?` does the same searching backward. Any address can be offset by a following `+`*n* or `-`*n*: `.+3` addresses three lines after the current one, and `/error/+1` addresses the line immediately following the next line containing "error."

13.3 Ranges

A range is two addresses separated by a comma, specifying a span from the first to the second, inclusive. `:5,10d` deletes lines 5 through 10. `:.,$d` deletes from the current line to the end of the buffer, combining two of the addresses introduced above. `%`, on its own, is shorthand for `1,$`, the entire buffer, and is consequently one of the most frequently used ranges in practice: `:%s/.../.../` (fully covered in Chapter 14) applies a substitution across every line in the file.

A range can also be constructed visually rather than typed by hand: selecting a span of lines in Visual mode and pressing `:` automatically populates the command line with `:'<,'>`, using two marks (`'<` and `'>`) that Vim sets automatically to the boundaries of the most recent Visual selection. This is worth knowing explicitly because it means every Ex command that accepts a range can be applied to a Visual selection without the reader needing to translate that selection into line numbers by hand; Vim performs that translation automatically the moment `:` is pressed while a selection is active.

A count following a range, rather than within it, has a distinct meaning worth flagging to avoid confusion: `:5,10d 3` does not multiply anything, the way a Normal-mode count might; it deletes 3 lines starting from line 10 (the end of the specified range), discarding the range's own start point in favor of using its end as a new starting line. This is a genuine irregularity in the grammar, inherited from ed and vi rather than a Vim invention, and is worth knowing exists primarily so that it is not stumbled into by accident.

13.4 Patterns as Addresses, Revisited

The pattern-as-address syntax introduced above deserves a second pass, because it is the mechanism behind one of Ex's most useful capabilities: locating and acting on lines by content rather than by position. `:/TODO/d` deletes the next line containing "TODO." More usefully still, patterns compose with ranges: `:/start/,/end/d` deletes every line from the next occurrence of "start" through the next subsequent occurrence of "end," a range defined entirely by content rather than by any line number the user had to look up or count. This pattern-defined range is the direct ancestor of the global command covered in Chapter 15, which generalizes "find lines matching a pattern" from a single range into a repeatable operation across every matching line in the buffer.

13.5 Command-Line History

Every Ex command typed is retained in a history, navigable while in Command-line mode with the up and down arrow keys (or, in a purely keyboard-centric workflow consistent with this book's general preference, `Ctrl-p` and `Ctrl-n`), recalling previous commands for re-execution or modification without retyping them from scratch. `q:` opens this history as an editable window, listing past commands as buffer lines that can be navigated, edited with the ordinary editing commands from Parts II and III, and executed directly with return, which is considerably more powerful than simple up-arrow recall when a long or complex previous command needs substantial modification rather than a single small correction. `:history` (or `:his`) displays the same history as plain output rather than an editable window, useful when only inspection, not re-execution, is needed. The read-only `":` register from Chapter 10, holding the single most recently executed Ex command, is the narrowest and most immediate slice of this same history.

13.6 Command-Line Completion

Pressing `Tab` while typing an Ex command triggers completion, context-sensitive to what is being typed: completing a filename after `:e` or `:w`, completing an option name after `:set`, completing a command name itself when only the command's leading characters have been typed. `Ctrl-d` lists all current completion candidates at once rather than cycling through them one at a time with repeated `Tab` presses, and `:set wildmenu` enables a visual, navigable completion menu along the bottom of the screen rather than the terser default behavior, generally a worthwhile addition to the minimal vimrc introduced in Chapter 3 for any reader who works with filenames or options frequently enough to benefit from seeing candidates listed rather than cycled through blindly.

13.7 Ex and Normal Mode as One Grammar

This chapter opened by asserting that Normal-mode keystrokes and Ex commands share an underlying grammar rather than constituting two separate systems, and the material covered since should make that assertion concrete rather than merely programmatic. An Ex address is, in a real sense, a motion expressed as typed text rather than as a keystroke: `.` and `$` mean the same thing whether pressed in Normal mode or typed on the Ex command line, and a search pattern addresses a line in Ex exactly as a search motion addresses a position in Normal mode. A range is, correspondingly, close kin to a text object: both specify a span for a following command to act upon, one expressed as two addresses, the other as a structural boundary. The remaining two chapters of Part V, on substitution and on the global command, are best read in this light: not as a separate skill from the operator-motion grammar of Part II, but as that same underlying grammar, continuing to generate new commands from a small set of parts, now expressed through a different, typed surface.

Chapter 14

Substitution

If any single Ex command justifies treating Ex as a language in its own right rather than a peripheral feature, it is `:s`. Substitution is where Vim's regular-expression pattern matching, Chapter 13's range syntax, and, at its most capable, arbitrary Vimscript expressions converge into a single command capable of transformations that would take many individual Normal-mode edits to accomplish by hand.

14.1 Basic Substitution

The general form is:

```
:[range]s/pattern/replacement/[flags]
```

With no range, `:s` operates on the current line only. `:s/foo/bar/` replaces the first occurrence of "foo" with "bar" on the current line. Combined with the range syntax from Chapter 13, `:%s/foo/bar/g` replaces every occurrence (the trailing `g` flag, covered below) of "foo" with "bar" across the entire buffer (`%`, the whole-buffer range), which is likely the single most common invocation pattern a reader will use in practice.

The delimiter need not be `/`. Any punctuation character following `s` is accepted as the delimiter for that invocation: `:s#foo#bar#` behaves identically to the slash-delimited form. This exists specifically to avoid awkward escaping when the pattern or replacement itself contains a literal `/`, such as a file path: `:s#/usr/local#/opt#` is considerably more readable than the same substitution with every literal slash escaped as `\/`.

14.2 Flags

`g` (global, not to be confused with the global command of Chapter 15, an unrelated feature that happens to share the same letter) replaces every match on each addressed line rather than only the first. Without `g`, `:s/o/0/` on a line containing "foo" produces "f0o"; with it, `:s/o/0/g` produces "f00." `c` requests confirmation before each individual replacement, prompting with the options to replace, skip, replace all remaining, or quit, which is worth reaching for whenever a substitution's scope is uncertain enough that blind, unconfirmed replacement across many lines feels risky. `i` and `I` force case-insensitive and case-sensitive matching respectively for that invocation only, overriding whatever the `ignorecase` option from Chapter 3 is currently set to.

14.3 Capture Groups

Parentheses in the pattern, escaped as `\(` and `\)` in Vim's default (non-`magic`) regex mode, capture the enclosed match for reuse in the replacement, referenced there as `\1`, `\2`, and so on for successive captured groups, with `\0` or `&` referring to the entire match. `:s/\(\w\+\)@\(\w\+\)/\2@\1/` swaps the text before and after an `@` sign, using two capture groups and referencing them in reversed order in the replacement.

14.4 Very Magic Mode

The escaping burden in the example just given, backslashes before nearly every metacharacter, is a consequence of Vim's default regex mode, in which most punctuation is treated literally unless explicitly escaped into its special meaning. Prefixing the pattern with `\v` switches to "very magic" mode for that pattern, in which parentheses, plus signs, pipes, and most other regex metacharacters behave as they would in most other regular-expression dialects, with no escaping required. The swap example above, rewritten very-magic: `:s/\v(\w+)@(\w+)/\2@\1/`, a meaningfully more readable pattern for anyone accustomed to regular expressions from another language or tool. A pattern collected from real editing use elsewhere in this book, matching lines beginning with "a" or "g," shows the same very-magic alternation syntax in a global-command context: `g/\v^(a|g)/d`, taken up fully in Chapter 15.

14.5 The Replacement Expression

Where the replacement text itself is not fixed but needs to be computed, prefixing it with `\=` evaluates the following text as a Vimscript expression rather than inserting it literally. `:s/\v(a|d|g)/\=strftime("%c")/` replaces each matched character with the current date and time, calling Vim's built-in `strftime` function from directly inside the substitution rather than requiring a separate step to compute and insert a timestamp. A related pattern replaces matched text with the contents of a register rather than a computed value: `:s/\v(fee)/\=@w/` replaces every match of "fee" with whatever register `w` currently holds, bridging the substitution command directly to the register material of Chapter 10. This is worth knowing exists even for a reader who rarely needs it, because it is the answer to a specific, recurring question, "how do I substitute in something I can't type directly into the pattern," that has no other clean solution within the ordinary literal-replacement syntax.

14.6 Worked Examples

The following substitutions are drawn from ordinary editing use rather than constructed for demonstration, and are worth working through individually, since each illustrates a distinct piece of the syntax covered above.

Normalizing punctuation. `:%s/---/--/g` replaces every em dash with a double hyphen across the buffer, a common normalization when preparing text for a context that does not render em dashes as intended. `:%s/\(\)/"/g` and its mirror for the closing curly

quote replace curly quotation marks with straight ones, handled as two separate substitutions since the opening and closing curly quotes are distinct characters with no single pattern matching both while leaving straight quotes alone.

Case normalization. `:%s/\u\+/\L&/g` matches runs of uppercase letters and lowercases them, using `\L` to case-convert everything through the next `\E` or the end of the replacement, combined with `&` to reference the whole match rather than retyping it. A more structurally aware variant capitalizes the first letter of every sentence: `:%s/\v(^|[.!?])\zs\w/\u&/g`, worth pausing on specifically for its use of `\zs` ("zero-width start"), which resets where the reported match begins without discarding what was matched before it. The pattern needs to match a sentence boundary (the start of the line, or a punctuation mark followed by a space) in order to correctly identify where a new sentence begins, but the replacement should touch only the following letter, not the boundary text itself; `\zs` is exactly the tool for that separation, and is worth remembering as the general solution whenever a pattern needs context to match correctly but a replacement that leaves that context untouched.

Removing structural noise. `:%s/\[.*\]//g` removes anything enclosed in square brackets, useful for stripping footnote or citation markers from a body of text; a more targeted variant, `:%s/\[\d*\]//g`, removes only numeric bracket markers such as `[1]`, leaving other bracketed content untouched. `:%s/^\s*//g` strips leading whitespace from every line; `:g/^\s*$/d` (a global-command deletion rather than a substitution, previewing Chapter 15) removes lines that are blank or contain only whitespace entirely, rather than merely stripping the whitespace and leaving an empty line behind. Collapsing three or more consecutive blank lines down to exactly one is a job for a substitution whose pattern and replacement both span multiple lines: `:%s/\n\n\n\+/\r\r/g`, using `!` as the delimiter to avoid escaping the pattern's own slashes is unnecessary here since none appear, but is worth remembering as an option; note that `\r` in the replacement denotes a literal newline, distinct from `\n`, which denotes a newline within the pattern being matched, an asymmetry inherited from Vim's underlying line-based buffer representation rather than an arbitrary inconsistency.

14.7 Substitution as the Ex Language's Center of Gravity

Nearly every capability introduced elsewhere in this book's treatment of Ex converges in this chapter: ranges from Chapter 13 restrict scope, registers from Chapter 10 supply computed replacement text, and, as Chapter 15 will show, the global command extends substitution's per-line matching into arbitrary per-line command execution. A reader comfortable with the material in this chapter has, in a real sense, already learned the hardest part of Ex; what remains in Part V is less about new syntax than about new ways of directing where substitution, and the commands it inspired, should apply.

Chapter 15

Global Editing

The pattern-as-address syntax from Chapter 13 located a single line by content. This chapter generalizes that idea from "find one matching line" to "find every matching line and act on each one," which is the last major piece of Ex's grammar and, for many experienced users, the single most powerful command Vim provides.

15.1 The Global Command

`:g/pattern/command` finds every line in the addressed range (the whole buffer, by default, if no range is given) matching *pattern*, and then executes *command* on each matching line in turn. This two-phase behavior, matching first across the whole range and only then executing, is worth understanding precisely, because it explains behavior that would otherwise be surprising: `:g/foo/d`, deleting every line containing "foo," correctly deletes every such line even though earlier deletions shift the line numbers of everything below them, because Vim identifies all matching lines before executing any command against them, rather than re-scanning after each deletion.

Without an explicit command, `:g/pattern/` defaults to `p` (print), listing matching lines, which is a quick way to preview what a global command would affect before committing to a more consequential one.

15.2 The Inverse Global Command

`:v/pattern/command` (an abbreviation of `:g!`) is the inverse: it executes *command* on every line that does *not* match *pattern*. `:v/^#/d`, applied to a shell script or configuration file, deletes every line that does not begin with a comment character, keeping only the comments, a targeted way to extract documentation from a file that intersperses it with executable content.

15.3 Worked Examples

Deleting matched lines. `:g/\v^(a|g)/d` deletes every line beginning with "a" or "g," using the very-magic alternation syntax from Chapter 14. `:g/^\d\+$/d` deletes every line consisting entirely of digits, useful for stripping standalone page numbers from a document that has been converted from a paginated format into continuous plain text.

Reversing a buffer. `:g/^/m0` reverses the order of every line in the buffer, and is worth working through slowly, since the mechanism is not obvious on first reading. `^` matches every line unconditionally (the start of the line, present everywhere). `m0` moves the matched line to a position immediately after line 0, meaning to the very top of the buffer. Because the global command processes matches in top-to-bottom order but each successive match is moved to the very top, the second line processed ends up above the first, the third ends up above the second, and so on; the cumulative effect, once every line has been processed, is a fully reversed buffer, achieved with no loop, no counter, and no auxiliary storage, purely as a side effect of repeatedly moving each matched line to the same fixed position.

Combining global with Normal-mode commands. `:g/\w$/normal $3X` finds every line ending in a word character and, on each, runs a Normal-mode command sequence: `$` to the end of the line, then `3X` to delete the three characters before the cursor. This is the global command's most generally useful capability, because it means every operator-motion command from Part II and every text object from Part II become, in effect, global-command arguments: anything expressible as a Normal-mode sequence can be applied to every line matching an arbitrary pattern, which multiplies the global command's own utility by the entire vocabulary covered in the first half of this book.

15.4 The `:normal` Command on Its Own

`:normal`, seen above as an argument to `:g`, is also a complete Ex command in its own right, executing the Normal-mode keystrokes that follow it as though they had been typed directly. `:5,10normal A;` appends a semicolon to the end of each line from 5 through 10, combining a range with a Normal-mode command exactly as `:g` combines a pattern match with one. `:normal` used without a preceding `:g` or explicit range applies only to the current line, which makes the combination with `:g` specifically, rather than `:normal` in isolation, the more commonly reached-for form in practice.

A note on syntax worth flagging explicitly: a literal `escape` keypress cannot simply be typed into an Ex command line the way ordinary characters can, since pressing `escape` while typing an Ex command cancels that command instead. Where a Normal-mode sequence given to `:normal` needs to include an escape (to end an Insert-mode segment embedded within it, for instance), it must be inserted literally into the command line with `Ctrl-v escape` rather than typed as escape directly, a small but easy-to-miss wrinkle for a reader constructing a more elaborate `:normal` invocation for the first time.

15.5 The `:execute` Command

`:execute` evaluates its argument as a Vimscript expression yielding a string, and then runs that string as an Ex command. This is what makes it possible to construct a command dynamically rather than typing it literally: `:execute "normal " . printf("%d", i) . "Gdd"`, inside a loop where `i` varies, deletes a different, computed line number on each iteration, something `:normal` alone cannot do since it treats its entire argument as literal keystrokes rather than as an expression to be evaluated first. A previewed instance of this

pattern, generating thousands of sequentially numbered lines, appears in Part VI's treatment of Vimscript, where `:execute` combines with a `:while` loop to insert content programmatically rather than by any single command covered in this chapter.

15.6 Buffer, Window, and Argument List Iteration

`:argdo`, `:bufdo`, and `:windo` generalize the global command's "do this to every matching line" pattern to a coarser unit: every file in the argument list, every open buffer, or every open window, respectively. `:argdo %s/old/new/g |update` applies a substitution across every file passed to Vim on the command line (the argument list, distinct from the buffer list of Chapter 12 in that it reflects what was explicitly specified at startup or via `:args`, rather than everything currently loaded), writing each modified file with `:update` (a variant of `:w` that writes only if the buffer was actually changed) before moving to the next. This is the mechanism behind applying a single substitution across an entire project's worth of files in one command, invoked either interactively or, as Chapter 19 shows, directly from the shell with `vim -c`.

15.7 Global Editing as the Ex Language Completed

This chapter closes Part V, and it is worth taking stock of what the three chapters together have established. Chapter 13 gave Ex its addresses and ranges; Chapter 14 gave it pattern-based transformation of text within a line; this chapter gives it pattern-based selection of which lines to act on at all, plus, through `:normal` and `:execute`, a bridge back to every capability covered in Parts II and III. The claim made at the start of Part V, that Normal mode and Ex are two surfaces of one grammar, should by this point be difficult to dispute: a global command that dispatches Normal-mode operator-motion sequences across every pattern-matched line in a file is not a separate skill bolted onto Vim's editing model, it is that editing model, addressed at a different scale.

Part VI

Automation

Chapter 16

Macros

Chapter 4 introduced the period command as a way of repeating the single most recent change. A macro generalizes that idea from one change to an arbitrary recorded sequence of them, and Part VI opens with it because a macro, more than anything else in this book, sits at the boundary between editing and programming: it is composed entirely of commands already covered in Parts II through V, yet using one well is closer to writing a small, disposable program than to issuing a single command.

16.1 Recording

`q`*letter*, pressed in Normal mode, begins recording into register *letter* (the same register namespace covered in Chapter 10); every keystroke from that point forward is recorded, verbatim, until `q` is pressed again to stop. `@`*letter* then replays the recorded sequence, and `@@` repeats whichever macro was most recently played, without needing to specify its register name again. A count prefixed to `@` replays the macro that many times in succession: `10@q` runs the macro in register `q` ten times.

What is recorded is not a semantic description of an edit but the literal keystroke sequence that produced it: an operator, a motion or text object, perhaps a search, perhaps an Insert-mode segment terminated by escape, exactly as it was typed. This is worth stating plainly because it explains both a macro's chief strength and its chief limitation. The strength is that a macro can capture anything expressible in Vim's own grammar, with no separate scripting syntax to learn beyond what Parts II through V have already covered. The limitation is that a macro replays the same keystrokes against whatever the cursor and buffer happen to look like at replay time, not the same literal effect: a macro recorded as "delete the word under the cursor" will delete whatever word the cursor happens to be on when replayed, which may or may not be the word the macro's author had in mind when validating it against the first instance.

16.2 Testing Before Scaling

The workflow that follows from this limitation, and that experienced Vim users converge on independently, is to test a macro cautiously before trusting it broadly. Record the transformation once, carefully, against a single representative instance; replay it once or twice more with `@q` to confirm it behaves as intended against a few more instances; and only once it has proven reliable, replay it at scale with a count, `100@q` or more, across the remainder of

the buffer. This is not merely caution for its own sake. A macro that fails partway through a large replay count typically does so because some instance among the targets differs structurally from the one the macro was recorded against, and catching that failure after two or three replays, while it is still easy to identify which instance broke the pattern, is considerably less costly than catching it after the macro has already run one hundred times and left the buffer in a partially, inconsistently transformed state.

This staged approach also gives a concrete, practical shape to a more general question worth asking before recording any macro at all: is this transformation going to be repeated often enough, across similar enough material, to be worth the overhead of recording and validating it in the first place. A transformation needed three times is very often faster to perform by hand three times than to record, test, and replay; a transformation needed fifty times, across material that is genuinely structurally consistent, is a clear case for a macro. The deciding factor is not the raw repetition count alone but repetition count weighed against how reliably the surrounding structure repeats: a macro applied against material with inconsistent formatting can cost more in cleanup after a failed run than it would have cost to simply perform the edits by hand from the start.

16.3 Nested and Recursive Macros

A macro recorded into one register can invoke another by including `@`*letter* as part of its own recorded keystrokes, which lets simple, single-purpose macros compose into more elaborate ones exactly as operators compose with motions in Part II: rather than recording one long, monolithic macro, several small macros, each handling one well-defined transformation, can be built and validated independently, then combined by having one invoke another.

A macro that invokes itself, by including `@q` as part of what is recorded into register `q`, becomes recursive, repeating until it encounters a condition under which the step it performs (frequently a search) fails, at which point the recursive call itself fails and the chain of replays stops. This is a genuine, if minimal, form of conditional looping, expressed entirely within Vim's own Normal-mode grammar rather than requiring the Vimscript control-flow constructs covered in Chapter 17: a recursive macro that searches for a pattern, transforms the found instance, and then invokes itself again will continue until no further instance of the pattern remains to be found, at which point the search fails, the recursive invocation fails along with it, and the macro terminates on its own.

16.4 Defining a Macro Without Recording It

A macro's contents are, ultimately, just the text stored in a register, which means a macro can be constructed directly with `:let @q = '...'` rather than performed live and captured. This is useful whenever the desired keystroke sequence is easier to reason about or generate programmatically than to physically type and record: a macro assembled from a computed string, or one that needs to embed a literal escape or control character that would be awkward to record by hand (using `\<Esc>` or similar notation within the assigned string to represent it), is often more reliably constructed this way than through live recording.

This is also the natural bridge into Chapter 17: a register assignment is itself a single line of Vimscript, and a macro built this way is, formally, no different from any other piece of scripted automation covered next.

16.5 Editing a Recorded Macro

Because a macro's contents are register text, they can be inspected and modified with the same tools that inspect and modify any other register content. `:let @q` (with no assignment) displays register `q`'s current contents in the command-line area, letting a recorded macro be reviewed after the fact. A more thorough editing workflow pastes the register's contents into the buffer as ordinary text (`"qp`, using the put command from Chapter 10), edits that text with any of the commands covered throughout Parts II and III, and then yanks the corrected line back into the register with `"qy$` rather than re-recording the entire macro from scratch. This is generally faster than live re-recording for anything beyond a trivial correction, and it is worth knowing as the default approach for fixing a macro that almost worked rather than one that failed outright.

16.6 The Pain-Point Threshold

The testing-before-scaling discipline described above is really an informal version of a more general question: given that recording, validating, and running a macro all cost time up front, at what point does that up-front cost pay for itself against simply performing the transformation by hand each time it is needed? Put concretely: if a manual repetition costs some amount of time and attention per instance, and recording a macro costs a roughly fixed amount of setup time regardless of how many instances it is eventually run against, then a macro is worth recording once the number of remaining instances is large enough that the fixed setup cost, spread across all of them, comes out lower than doing each one by hand. For three or four remaining instances, that threshold is very often not met, and manual editing remains the faster choice; for fifty structurally consistent instances, it very clearly is. Reasoning explicitly in these terms, rather than reaching for a macro reflexively the moment any repetition appears, is worth doing consciously until it becomes second nature, since a macro recorded (and then debugged, when it inevitably breaks on some unanticipated variation) against only three or four remaining instances frequently ends up costing more total time than it saved.

This same reasoning has a second edge worth stating explicitly: the threshold above assumes the repeated material is genuinely structurally consistent. A macro recorded against material that only looks consistent, but in fact varies in some way the macro's author did not anticipate, will fail partway through a large replay, and the cost of noticing the failure, diagnosing what varied, and cleaning up whatever the macro did to the instances it mishandled can easily exceed whatever time the macro saved on the instances it handled correctly. This is the deeper reason behind the testing discipline from earlier in this chapter: validating against a few instances before scaling up is not merely a way to catch an outright broken macro, it is a way of empirically checking whether the structural consistency the whole cost-benefit calculation depends on actually holds, before committing to it at scale.

16.7 Macro Libraries

A macro recorded during one editing session does not persist to the next unless deliberately saved, since registers, like the undo history covered in Chapter 9, are session-local by default. Where a particular transformation is needed repeatedly across many separate sessions, rather than repeatedly within a single one, the more durable solution is to store its `:let @q = '...'` definition in the vimrc introduced in Chapter 3, populating register `q` (or whichever register is chosen) automatically every time Vim starts. A reader who finds themselves recording substantially the same macro in separate sessions, days or weeks apart, has outgrown live recording for that particular transformation and should consider promoting it into the vimrc as a standing, always-available macro instead, which is a small-scale instance of the same promotion-from-manual-to-persistent-automation pattern that motivates moving from a macro to a full Vimscript function, the subject of the next chapter, once a transformation's complexity outgrows what a macro can comfortably express.

Chapter 17

Vimscript

A macro, per Chapter 16, is a recorded sequence of Vim's own commands, useful precisely because it requires no vocabulary beyond what Parts II through V already cover. Vimscript is what a reader reaches for once a transformation's logic genuinely exceeds what a linear sequence of recorded keystrokes can express: conditional behavior, named and reusable procedures, and configuration that reacts to events rather than being invoked directly. This chapter is not a complete language reference; it covers the fraction of Vimscript that a reader of this book is most likely to need, and points toward `:help` for anything further, consistent with Chapter 2's treatment of the help system as a self-sufficient resource in its own right.

17.1 Variables

`:let x = 5` assigns the value 5 to a variable named `x`, and `:echo x` displays it. Variable names are, by default, scoped to whatever context they are defined in (a script, a function), but an explicit prefix narrows or widens that scope deliberately: `g:` for a global variable, visible everywhere; `s:` for one local to the current script file; `b:` for one local to the current buffer, useful for state that should differ file by file rather than being shared across an entire session. The registers already covered in Chapter 10 are, from Vimscript's perspective, themselves a kind of variable, addressed with the `@` prefix rather than assigned a name: `:let @q = 'daw'` sets register `q`'s contents directly, exactly as seen in the previous chapter's treatment of constructing a macro without recording it.

17.2 Functions

```
function! DoubleIndent()
   normal! >>
   normal! >>
endfunction
```

`function!` defines a function, with the trailing `!` permitting redefinition without error if a function of the same name already exists, generally desirable while iterating on a definition during a single editing session. `normal!` inside a function behaves like the `:normal` command from Chapter 15, executing Normal-mode keystrokes, with the `!` suppressing any user-defined mappings that might otherwise interfere with keystrokes intended literally. `:call DoubleIndent()` invokes the function. A function that returns a value does so

with an explicit `return` statement, and functions intended to operate on a range, callable as `:'<,'>call MyFunction()` from a Visual selection in the manner of Chapter 13, are declared with a trailing `range` keyword so that Vim passes the selection's boundaries in automatically as the built-in `a:firstline` and `a:lastline` variables.

17.3 Conditionals

```
if line('.') == 1
    echo "first line"
elseif line('.') == line('$')
    echo "last line"
else
    echo "middle"
endif
```

Vimscript's `if`/`elseif`/`else`/`endif` behaves as its syntax suggests, and `line('.')` and `line('$')`, used here, are built-in functions returning the current line number and the buffer's last line number respectively, callable anywhere a Vimscript expression is valid, including inside the \= replacement-expression syntax from Chapter 14.

17.4 Loops

```
let i = 1
while i <= 10
    execute 'normal! I' . i . '. '
    normal! j
    let i += 1
endwhile
```

This loop prefixes each of the first ten lines with its own number and a period, using `execute` (from Chapter 15) to build a dynamic Normal-mode command string on each iteration and `normal!` to move down one line between iterations. A structurally similar pattern, encountered as real, previously-used automation rather than a constructed example, generates several thousand sequentially numbered lines in a single invocation:

```
let i = 1
while i <= 18000
    execute 'normal! i' . printf("fr/fr_%05d.mp3", i)
    let i += 1
endwhile
```

The two examples differ only in what each iteration inserts, one case using `printf` to zero-pad a number into a five-digit filename pattern rather than a bare line number; the loop structure itself, an incrementing counter and a `while` condition, is identical. This is worth noting explicitly because it is easy to mistake a large iteration count for a fundamentally more complex piece of code than a small one; scaling a loop from ten iterations to eighteen thousand changes nothing about its structure, only the bound it is checked against.

17.5 Mappings

A mapping binds a key sequence to a command, extending the built-in Normal-mode vocabulary of Part II with new, custom key sequences of the reader's own choosing. `nnoremap <C-s> <C-a>h` binds Control-s, in Normal mode, to the built-in increment command `Ctrl-a` followed by `h`. The trailing `h` is not incidental: `Ctrl-a` increments the number under or after the cursor and leaves the cursor positioned at the end of the changed number, so the following `h` moves it back by one character, a small correction that keeps the cursor's resting position consistent with where it would have been had no increment occurred.

The `nore` in `nnoremap` (as opposed to plain `nmap`) specifies a non-recursive mapping: the right-hand side, `<C-a>h`, is interpreted using Vim's built-in meanings for those keys, not through any further layer of the user's own mappings that might otherwise apply to the same keys. This distinction matters increasingly as a reader's vimrc accumulates more mappings over time, since a recursive mapping (plain `nmap`) can interact unpredictably with other mappings defined later, while a non-recursive one (`nnoremap`) cannot; for this reason, `nnoremap` and its counterparts for other modes (`inoremap` for Insert mode, `vnoremap` for Visual mode, and so on) are the generally recommended default, with plain, recursive mapping reserved for the specific, less common cases where recursion is actually intended.

17.6 Autocommands

An autocommand binds a piece of Vimscript to an event, executing automatically whenever that event occurs rather than requiring an explicit invocation. `autocmd BufWritePre *.py normal! gg=G` runs the `gg=G` auto-indent command (using the `=` operator from Chapter 7 across the entire buffer, from the `gg` motion to the `G` motion of Chapter 5) immediately before writing any file matching the pattern `*.py`, keeping Python files consistently indented without the user having to remember to trigger the indentation manually each time. `BufWritePre` is one of a large set of defined events (others include `BufReadPost`, triggered after a file is loaded; `FileType`, triggered when a buffer's detected file type is set; and `VimEnter`, triggered once at startup), each documented in full under `:help autocmd-events`. Autocommands are generally grouped with `augroup`, paired with an explicit `autocmd!` to clear any previous definitions in that group, to prevent the same autocommand from being registered multiple times if a vimrc is re-sourced during a session, a subtlety worth knowing before a reader's first autocommand mysteriously appears to run twice.

17.7 Vimscript as the Ceiling of This Book's Automation Material

This chapter, together with the macros of Chapter 16, completes Part VI, and it is worth being explicit about where this book's coverage of automation ends. Vimscript is a complete, Turing-complete scripting language with a considerably larger surface than the variables, functions, conditionals, loops, mappings, and autocommands covered here: dictionaries and lists as first-class data structures, a module system for larger plugins, and integration with other scripting languages Vim can be compiled against, among a great deal else documented under `:help usr_41.txt`, Vim's own guide to writing Vimscript. This book's

plugin-free scope, stated in the Preface, means it stops at the point where a reader has enough Vimscript to automate their own editing rather than to build distributable tooling for others; a reader who finds themselves wanting more than this chapter provides has, in a real sense, outgrown this book's remit for that specific need, and `:help usr_41.txt` is the appropriate next stop rather than a shortcoming of what is covered here.

Part VII

Unix Integration

Chapter 18

Shell Commands

Chapter 7 introduced `!` as an operator that sends spanned text to an external command and replaces it with that command's output, previewing a theme this chapter now develops fully: Vim does not treat the shell as something outside itself to be escaped to occasionally, but as a resource woven throughout its own command language. Part VII is where this book's plugin-free, Unix-native scope pays off most directly, and this chapter is its foundation.

18.1 Executing a Command

`:!`*command* runs *command* in a shell and displays its output, without touching the buffer at all. `:! ls` lists the current directory's contents; `:!!` repeats the most recently executed shell command. This is the simplest point of contact between Vim and the shell, useful whenever a quick check (does this file exist, what does this directory contain, did that background process finish) is needed without leaving the editor or losing the current buffer's state.

18.2 Reading Command Output Into the Buffer

`:r !`*command* runs *command* and inserts its output into the buffer after the current line, exactly as `:r` *filename* (from Chapter 8) reads a file's contents, except that the source is a command's output rather than a file's. `:r !ls` inserts a directory listing directly into the buffer at the cursor, a genuinely common move when drafting a document that needs to enumerate files, or when bootstrapping a data file from some external source's output rather than typing that data by hand.

18.3 Filtering the Buffer Through a Command

`:%!`*command* pipes the entire buffer through *command* and replaces the buffer's contents with that command's output; `:'<,'>!`*command*, following a Visual selection per the range syntax of Chapter 13, does the same for only the selected lines. `:%!sort` sorts every line in the buffer; `:'<,'>!column -t`, applied to a selection of whitespace-separated data, reformats it into aligned columns. This is the Ex-command form of the same operation the Normal-mode `!` operator from Chapter 7 performs against a motion or text object; `!}sort`, from that chapter, and `:.,'a!sort`, expressed as an Ex range, accomplish comparable ends through the two different surfaces of Vim's grammar already unified in Chapter 13.

The distinction between `:r !` and `:%!` is worth holding onto precisely: the former adds a command's output to the buffer without removing anything, while the latter replaces buffer content with a command's output, using the existing content as that command's input. `:r !ls` and `:%!sort` are doing structurally different things even though both involve a shell command and a colon, and conflating them is a common early mistake.

18.4 Sending the Buffer to a Command

`:w !`*command* sends the current buffer's content to *command* as standard input, displaying whatever that command prints in response, without modifying the buffer itself, distinct from `:%!`, which does modify it. `:w !python3` runs the current buffer's content through a Python interpreter and shows the result, a fast way to test a short script without saving it to a file first; `:w !grep pattern` shows which lines of the current, unsaved buffer would match a pattern, without needing to save first and run `grep` against the file from a separate shell.

18.5 Working Directory and Sub-Shells

`:cd` *directory* changes Vim's own working directory, which affects how relative paths in subsequent commands, including all the shell-interaction commands covered in this chapter, are resolved. `:sh` starts an interactive sub-shell, suspending Vim and dropping into an ordinary shell prompt within the same terminal; `Ctrl-d` (or `exit`) from that sub-shell returns control to Vim exactly where it was left, buffers and all. This is a coarser tool than the command-specific forms above, useful when a genuinely open-ended shell session is needed rather than a single command's output, and it is worth knowing as an option precisely because it means a reader is never truly boxed into Vim's own command surface; the full shell is always one command away.

18.6 A Note on Workflow

The commands in this chapter make it possible to accomplish quite a lot without ever leaving Vim, and a reader newly introduced to them may be tempted to route everything through `:!`, `:r !`, and `:%!` on principle. This is worth resisting where a more direct alternative already exists elsewhere in this book, and worth embracing where it genuinely does not. Selecting text with the mouse and copying it through a terminal emulator's own clipboard handling, rather than through Vim's registers, remains the more convenient path for moving text into an entirely different application outside the terminal altogether, as already noted in Chapter 18's predecessor material in Chapter 10; the commands covered here are at their best when the task is fundamentally about transforming a buffer's content using an external tool's capability, not when it is about moving text between applications that have perfectly good clipboard integration of their own.

18.7 Shell Integration as a Pipeline Stage

It is worth closing this chapter by naming explicitly what the four core commands above have in common, since it is easy to see them as four unrelated features rather than one coherent capability. `:!`, `:r !`, `:%!`, and `:w !` are, respectively: run a command and discard the buffer's relationship to it entirely; bring a command's output in; send the buffer out, replacing it with the result; and send the buffer out, without replacing it. Between them, every direction of the relationship between "the buffer" and "an external process" is covered, in and out, with and without modifying the buffer as a side effect. Vim, in this light, is not an island that occasionally makes an exception to talk to the shell; for the duration of any of these four commands, it is one stage in a pipeline, receiving input from an upstream process or sending output to a downstream one, with a human editor positioned at exactly the point in that pipeline where judgment or manual correction is most needed. The chapters that follow, on development workflows and on writing, are largely an exploration of what becomes possible once that framing is taken seriously.

Chapter 19

Development Workflows

Chapter 3 noted, in passing, that Vim is Git's default editor: the program invoked automatically whenever Git needs a human to write or review something it cannot resolve on its own. This chapter takes that fact seriously and works through what it actually means in practice, since a substantial fraction of a working programmer's contact with Vim happens exactly here, often before they have deliberately chosen to learn the editor at all.

19.1 Commit Messages

When `git commit` is run without `-m`, Git opens the configured editor on a temporary file, pre-populated with a commented template, and expects a commit message written and saved back to that file, closing the editor to complete the commit or leaving the message empty (or the file unsaved, depending on configuration) to abort it. This file is an ordinary Vim buffer in every respect covered so far in this book: Insert mode types the message, Normal-mode commands from Parts II and III edit it, and `:wq` from Chapter 3 saves and exits to complete the commit. The lines beginning with `#` are comments, stripped automatically by Git before the message is recorded, and are generally left alone rather than deleted, since they typically show which files are staged and provide useful confirmation of exactly what the commit is about to include.

19.2 Interactive Rebase

`git rebase -i` *commit* opens a considerably more structured buffer: one line per commit being rebased, each beginning with an action word, `pick` by default, followed by that commit's abbreviated hash and message. This file is not merely a message to be written; it is a plan, and editing it edits what Git is about to do to the repository's history.

The most common edit, changing several `pick` lines to `squash` (folding a commit into the one before it) or its abbreviated form `s`, is an excellent worked example of the repeat command from Chapter 4 applied to real, structured work rather than to prose. Position the cursor on the word `pick` on the first line to be squashed and press `cw`, the change-word command from Chapter 7, type `squash`, and return to Normal mode. Move to the next `pick` line needing the same treatment and press `.`: because `cw` plus the typed replacement constitutes a single recorded change, per the compound-operator behavior of `c` noted in Chapter 7, the period command replays the entire substitution, word and replacement together, against the new line. A rebase touching many commits becomes a short, rhythmic

sequence: move to the next line, return to its first column with `0` from Chapter 5, press `.`, and repeat, considerably faster and less error-prone than manually retyping `squash` on each line by hand.

Reordering commits is equally direct: `dd` removes the current line into a register exactly as it would in any other buffer, per Chapter 7, and `p` or `P` places it below or above the cursor at its new position. Nothing about this is metaphorical or specific to rebase files; it is the same line-oriented deletion and placement covered throughout Part III, applied here to a file whose lines happen to represent commits rather than prose.

19.3 Merge Conflicts

When Git cannot automatically reconcile two branches' changes to the same region of a file, it leaves that file with the conflict marked directly in its text, delimited by `<<<<<<<`, `=======`, and `>>>>>>>`. Opening such a file in Vim presents the conflict as exactly what it is: ordinary text, with unusual but perfectly regular structural markers, resolvable with the same commands covered throughout this book rather than any conflict-specific vocabulary. A search for `<<<<<<<` with `/` from Chapter 5 navigates directly to the next conflict; the operators and text objects of Parts II and III delete whichever side of the conflict is being discarded, along with the marker lines themselves, leaving a single, coherent resolution behind.

A pattern worth knowing specifically for this task: `:g/^<<<<<<</,/^=======/d` deletes every line from a conflict's opening marker through its middle marker, discarding the current branch's version of a conflicted section in a single command, if the incoming branch's version is the one to keep; the corresponding deletion from the middle marker through the closing one discards the incoming version instead. Constructing this command is a direct application of the pattern-defined-range technique from Chapter 13, combined with the deletion command from `:g`, exactly as covered in Chapter 15, and is considerably faster than manually selecting and deleting each side of a conflict by hand when a file contains many conflicted sections following the same resolution pattern.

19.4 Comparing Files with `vimdiff`

`vimdiff` *file1 file2* (or, equivalently, `vim -d`) opens two or more files side by side in vertically split windows, per Chapter 12, with differing regions highlighted directly in the buffer text. `]c` and `[c` jump to the next and previous difference; `do` (diff obtain) pulls the other window's version of the current difference into the current window, and `dp` (diff put) pushes the current window's version to the other. Git can be configured to use `vimdiff` as its merge tool directly (`git config merge.tool vimdiff`), at which point `git mergetool` opens conflicts in this side-by-side view rather than as inline markers within a single buffer, a genuinely different and, for some readers, more legible way to work through the same underlying conflict-resolution task covered above.

19.5 The Quickfix List and `make`

`:make` runs the project's configured build command (by default, simply `make`, though the `makeprg` option can point it at any other build tool) and captures its output into the quickfix list: a structured list of file-and-line-number references, generally corresponding to compiler errors or warnings. `:copen` opens the quickfix list in its own window, one entry per line, and pressing return on any entry jumps directly to the corresponding file and line; `:cnext` and `:cprevious` step through the list without needing to return to the quickfix window each time. This transforms the otherwise manual work of reading a build's error output and separately navigating to each reported location into a single, structured workflow, staying entirely within Vim rather than switching between a terminal's scrollback and the editor.

19.6 Searching a Project with `grep`

`:grep` *pattern* runs an external `grep` (or, on some systems, a faster alternative it has been configured to use instead) across the project and populates the quickfix list with every matching line, exactly as `:make` does with build errors; the same `:copen`, `:cnext`, and `:cprevious` commands then navigate the results. A closely related built-in alternative, `:vimgrep` *pattern* `**/*.py`, searches using Vim's own regex engine rather than shelling out to an external program, which trades some speed on very large codebases for exact consistency with the regex syntax already covered in Chapter 14, since a pattern that works in `:s` is guaranteed to work identically in `:vimgrep`, which is not always true of an external `grep`'s own regex dialect.

19.7 Editing History as Editing Text

The material in this chapter has one idea running underneath all of it, worth stating explicitly now that the individual workflows have been covered: a commit message, a rebase plan, a conflicted file, and a build's error output are all, structurally, just text, and every one of them yields to the same grammar covered throughout this book rather than requiring any Git-specific or build-specific vocabulary of its own. The apparent gap between "editing a document" and "managing a project's history" turns out not to be a gap in kind at all, only in what the text being edited happens to represent. A reader fluent in the material of Parts II through V is already fluent in everything this chapter has covered; what this chapter has added is only the recognition of where that fluency applies.

Chapter 20

Writing

This book has drawn most of its examples from source code and structured data, consistent with Vim's reputation as a programmer's editor, but nothing in the grammar covered throughout Parts II through VI is specific to code. This closing chapter of Part VII turns to long-form prose: Markdown, LaTeX, and the particular demands of a manuscript large enough to span many files and many editing sessions, the kind of writing this book itself was produced with.

20.1 Markdown

Markdown's structural elements, headings, emphasis markers, links, and code spans, are plain text with light punctuation conventions, which means they are addressable with the ordinary motions and text objects of Part II rather than requiring any Markdown-specific commands. A heading is a line beginning with one or more `#` characters, navigable with the paragraph and section motions of Chapter 5 depending on how a document is structured; emphasis markers (`*`, `**`, `_`) delimit a span exactly as quotes do, and while Vim has no dedicated built-in text object for "the word between asterisks," the quote text objects from Chapter 6, `i*` is not built in, though a reader who finds themselves wanting it is precisely the audience for the user-defined text objects noted at the end of that chapter.

The formatting operators `gq` and `gw` from Chapter 7 are particularly suited to Markdown prose, rewrapping a paragraph to the configured `textwidth` after an edit has left its line lengths ragged; setting `textwidth=80` (or whatever width a particular publishing target expects) and reflowing edited paragraphs with `gqip` as a matter of habit keeps a Markdown document's source consistently formatted even under heavy revision, which matters more than it might first appear, since a version-controlled document with inconsistent line wrapping produces much noisier diffs than one with consistent wrapping, obscuring genuine content changes among incidental rewrapping in every diff review.

20.2 LaTeX

LaTeX introduces genuine structural complexity beyond Markdown's light punctuation, and several of Vim's built-in facilities apply to it directly and usefully. The tag text object from Chapter 6, `it`/`at`, is defined for markup with paired opening and closing delimiters and does not, out of the box, recognize LaTeX's `\begin{...}` / `\end{...}` environment syntax the way it recognizes HTML or XML tags; a reader working in LaTeX heavily enough

to want an equivalent should expect to reach for a filetype-specific plugin or a custom text object rather than the built-in `it`/`at` directly, consistent with this book's plugin-free scope simply noting where a natural extension point exists rather than claiming built-in coverage that isn't there. Percent-delimited comments, section and subsection commands, and citation keys, by contrast, are all addressable with the ordinary word and WORD motions of Chapter 5 without any special handling at all, since they are simply punctuation-delimited text like any other.

`:set spell` enables spell-checking directly in the buffer, underlining suspect words without disrupting the surrounding LaTeX markup, and `]s` and `[s` jump to the next and previous flagged word; `z=` at a flagged word suggests corrections, and `zg` adds a word to the session's known-good word list, useful for the field-specific terminology and citation keys that inevitably appear throughout a technical manuscript and would otherwise be flagged repeatedly.

20.3 Large Manuscripts

A book-length manuscript, split across many files (one per chapter, in the manner this very book's own source is organized), benefits from several of the multi-file mechanisms covered earlier in this book, brought together here for the specific case of long-form writing. The argument list from Chapter 15, populated with `:args chapters/*.tex` or a similar glob, makes `:argdo` available across every chapter file at once: a terminology change, a formatting correction, or any of the substitution patterns from Chapter 14 can be applied uniformly across an entire manuscript in a single command, exactly as Chapter 15 demonstrated for smaller-scale editing tasks.

`:vimgrep` *pattern* `chapters/*.tex`, from Chapter 19, searches across every chapter file for a pattern, populating the quickfix list with every match, which is the practical way to answer a question like "which chapters mention this term" or "where did I leave that unfinished cross-reference" across a manuscript too large to hold in memory or search file by file. Global marks from Chapter 11 are worth setting deliberately at points a writer expects to return to repeatedly, such as an outline file or a running notes document, since they remain addressable from within any chapter file without first navigating there manually.

Finally, a session, from Chapter 12, is the natural unit for a long-running writing project specifically: saving the exact arrangement of open chapter files, splits, and cursor positions at the end of a working session and restoring it at the start of the next removes the friction of re-opening and re-navigating to the same working set of files every time work resumes, which matters more for a manuscript revisited over weeks or months than for almost any other kind of editing task covered in this book.

20.4 Notes and Documentation

Plain-text notes, whether a single running file or a directory of many small ones, benefit from a lighter version of the same large-manuscript machinery. Marks and the jump list from Chapter 11 suffice for a single large notes file; for a directory of many small ones, `:e` combined with the filename completion from Chapter 13 is frequently faster than any

more elaborate navigation scheme, particularly once note filenames follow a consistent, predictable convention that completion can exploit. The global command from Chapter 15, applied across a set of notes opened via the argument list, answers questions like "which notes mention this project" exactly as it would across a manuscript's chapters, since a directory of notes and a book's chapter files are, from Vim's perspective, the same kind of thing: a set of plain-text buffers addressable together.

20.5 Writing as the Terminus of This Book's Argument

This chapter closes Part VII, and with it every applied chapter this book has to offer before Part VIII's reference material begins. It is worth noting, in closing, what has and has not been true throughout this chapter: not one command introduced here is new. Every mechanism this chapter has applied to Markdown, LaTeX, and large manuscripts was already established, for other purposes, somewhere in Parts II through VI. This is not a coincidence and not a limitation; it is the entire thesis of this book, restated one final time in its most human-facing context. A reader who arrived at this chapter already fluent in the grammar covered earlier did not need a separate vocabulary for writing prose, because Vim does not have one. Text is text, whether it is a function definition, a rebase plan, a merge conflict, or a paragraph of English prose, and the same small set of primitives, composed the same small number of ways, was sufficient for all of it.

Part VIII

Reference

Using This Reference

Parts I through VII taught Vim's grammar from first principles, deriving each command as an instance of a small number of composable rules rather than presenting commands as a list to memorize. Part VIII inverts that approach deliberately. What follows is the list: every normal-mode command, every Ex command, every motion, operator, text object, register, option, built-in function, regular-expression construct, and command-line argument covered or implied by the preceding chapters, organized as tables for lookup rather than as prose for reading start to finish.

This inversion is intentional rather than a lapse in the book's own stated method. A reference table serves a different purpose than a derivation: it answers "what does this do" or "what was that command called again" in the few seconds such a question deserves, without requiring the reader to re-read the chapter that originally introduced it. Each appendix notes, where relevant, which chapter treats the item in question at greater depth, so that a table entry can serve as an index back into the derivation as easily as it serves as a standalone answer.

Nothing in this part introduces new material. Every command listed here was already covered, directly or as an immediate consequence of the grammar, somewhere in Parts I through VII. A reader who has worked through those parts in order will find Part VIII entirely familiar; a reader using this book primarily as a desk reference, consulting Part VIII directly without having read the derivation first, will find each table sufficient on its own for looking up a specific command's syntax, at the cost of the deeper why that the earlier parts were written to provide.

Appendix A

Every Normal-Mode Command

Organized by category, matching the order these commands were introduced throughout Parts II and III. Within each category, commands are listed roughly in the order a reader would find most natural to learn them, not alphabetically. A count prefix (`[count]`) is omitted from the Command column wherever it applies generally, per the grammar established in Chapter 4.

A.1 Character and Line Movement

Command	Effect	Ch.
`h`	Move left one character	5
`l`	Move right one character	5
`j`	Move down one line	5
`k`	Move up one line	5
`0`	First column of current line	5
`^`	First non-blank character of current line	5
`$`	Last character of current line	5
`gg`	First line of buffer	5
`G`	Last line of buffer (or line `[count]`)	5
`+` / `return`	First non-blank of next line	5
`-`	First non-blank of previous line	5

A.2 Word, Sentence, and Paragraph Movement

Command	Effect	Ch.
`w`	Beginning of next word	5
`W`	Beginning of next WORD (whitespace-delimited)	5
`b`	Beginning of previous word	5
`B`	Beginning of previous WORD	5
`e`	End of current/next word	5
`E`	End of current/next WORD	5
`ge`	End of previous word	5
`gE`	End of previous WORD	5

Command	Effect	Ch.
`(`	Beginning of current/previous sentence	5
`)`	Beginning of next sentence	5
`{`	Beginning of current/previous paragraph	5
`}`	Beginning of next paragraph	5
`[[`	Beginning of current/previous section	5
`]]`	Beginning of next section	5

A.3 Search and Character-Find Movement

Command	Effect	Ch.
`/pattern`	Search forward	5
`?pattern`	Search backward	5
`n`	Repeat search, same direction	5
`N`	Repeat search, opposite direction	5
`f`*char*	Find *char* forward on line, land on it	5
`F`*char*	Find *char* backward on line, land on it	5
`t`*char*	Find *char* forward, land before it	5
`T`*char*	Find *char* backward, land after it	5
`;`	Repeat last `f/F/t/T`	4
`,`	Repeat last `f/F/t/T`, reversed	4

A.4 Screen and Absolute Position

Command	Effect	Ch.
`H`	Top line of screen	5
`M`	Middle line of screen	5
`L`	Bottom line of screen	5
`[count]G`	Go to line `[count]`	5
`[count]|`	Go to column `[count]` of current line	5
`'`*mark*	Go to line of *mark*	11
`` ` ``*mark*	Go to exact position of *mark*	11
`` `` ```	Toggle to position before last jump	11

A.5 Operators

Command	Effect	Ch.
`d`{motion/object}	Delete	7
`c`{motion/object}	Change (delete, enter Insert)	7

Command	Effect	Ch.
`y`{motion/object}	Yank	7
`gU`{motion/object}	Uppercase	7
`gu`{motion/object}	Lowercase	7
`g~`{motion/object}	Toggle case	7
`>`{motion/object}	Indent right one shift width	7
`<`{motion/object}	Indent left one shift width	7
`=`{motion/object}	Auto-indent (filetype rules)	7
`!`{motion/object}	Filter through shell command	7
`gq`{motion/object}	Format (wrap to `textwidth`)	7
`gw`{motion/object}	Format, preserving cursor position	7
`dd`, `cc`, `yy`, `>>`, `<<`, `guu`, `gUU`	Doubled form: operate on current line	7

A.6 Inserting and Replacing

Command	Effect	Ch.
`i`	Insert before cursor	8
`a`	Insert (append) after cursor	8
`I`	Insert at first non-blank of line	8
`A`	Append at end of line	8
`o`	Open new line below, insert	8
`O`	Open new line above, insert	8
`R`	Enter Replace mode	8
`gR`	Enter Virtual Replace mode	8
`r`*char*	Replace one character with *char*	8
`s`	Substitute character (delete + Insert)	8
`S`	Substitute line (delete content + Insert)	8
`J`	Join next line, one space	7
`gJ`	Join next line, no space	7

A.7 Deletion and Change Shorthand

Command	Effect	Ch.
`x`	Delete character under cursor	7
`X`	Delete character before cursor	7
`D`	Delete to end of line (`d$`)	7
`C`	Change to end of line (`c$`)	7
`Y`	Yank current line (`yy`)	7

A.8 Put, Undo, and Repeat

Command	Effect	Ch.
`p`	Put after cursor/line	10
`P`	Put before cursor/line	10
`u`	Undo	9
`Ctrl-r`	Redo	9
`U`	Undo all changes to current line (toggle)	9
`g-`	Step backward through undo history by time	9
`g+`	Step forward through undo history by time	9
`.`	Repeat last change	4

A.9 Marks and Registers

Command	Effect	Ch.
`m`*letter*	Set mark *letter*	11
`"`*reg*	Prefix: address register *reg* for next op	10
`g;`	Previous position in change list	11
`g,`	Next position in change list	11
`Ctrl-o`	Older position in jump list	11
`Ctrl-i`	Newer position in jump list	11
`Ctrl-^`	Switch to alternate file	11

A.10 Macros

Command	Effect	Ch.
`q`*letter*	Begin/end recording macro into register *letter*	16
`@`*letter*	Play macro from register *letter*	16
`@@`	Repeat most recently played macro	16

A.11 Visual Mode

Command	Effect	Ch.
`v`	Enter character-wise Visual mode	2
`V`	Enter line-wise Visual mode	2
`Ctrl-v`	Enter block-wise Visual mode	6
`gv`	Reselect last Visual selection	6
`o`	(in Visual mode) Swap cursor to other end of selection	6

A.12 Windows and Tabs

Command	Effect	Ch.
`Ctrl-w s`	Split window horizontally	12
`Ctrl-w v`	Split window vertically	12
`Ctrl-w` *h/j/k/l*	Move to window in that direction	12
`Ctrl-w w`	Cycle to next window	12
`Ctrl-w c`	Close current window	12
`Ctrl-w =`	Equalize window sizes	12
`Ctrl-w r`	Rotate windows	12
`Ctrl-w x`	Exchange window with next	12
`gt`	Next tab	12
`gT`	Previous tab	12

A.13 Miscellaneous

Command	Effect	Ch.
`Ctrl-a`	Increment number under/after cursor	17
`Ctrl-x`	Decrement number under/after cursor	17
`Ctrl-g`	Show file name and position status	2
`Ctrl-k` *code*	Insert digraph	8
`Ctrl-v` *code*	Insert character literally by code point	8
`Ctrl-n`	(Insert mode) Keyword completion, next	8
`Ctrl-p`	(Insert mode) Keyword completion, previous	8
`]s` / `[s`	Next/previous misspelled word (with `spell`)	20
`z=`	Suggest spelling corrections	20
`]c` / `[c`	Next/previous diff (in `vimdiff`)	19
`do` / `dp`	Diff obtain / diff put	19

Appendix B

Every Ex Command

Every Ex command covered in this book, in the form typed on the command line (a leading `:` is implied throughout and omitted from the table for brevity). Ranges, where a command accepts one, follow the syntax of Chapter 13 and are omitted from the Command column unless a specific range usage is idiomatic enough to be worth showing directly.

B.1 File and Session Management

Command	Effect	Ch.
`w`	Write buffer	3
`w` *file*	Write buffer to *file*	3
`wq` / `x`	Write and quit	3
`q`	Quit	3
`q!`	Quit, discarding changes	3
`e` *file*	Edit *file*	2
`e!`	Reload current file, discarding changes	3
`r` *file*	Read *file* in after current line	8
`mksession` *file*	Save window/tab/buffer arrangement	12
`source` *file* (`:so`)	Execute commands in *file*	3

B.2 Buffers, Windows, and Tabs

Command	Effect	Ch.
`ls` / `buffers`	List all buffers	12
`b`*n*	Switch to buffer *n*	12
`b`*name*	Switch to buffer matching *name*	12
`bnext` (`:bn`)	Next buffer	12
`bprevious` (`:bp`)	Previous buffer	12
`bd`	Delete (unload) buffer	12
`split` (`:sp`)	Split window horizontally	12
`vsplit` (`:vsp`)	Split window vertically	12
`tabnew`	Open new tab	12
`tabclose`	Close current tab	12

Command	Effect	Ch.
`tabonly`	Close all tabs but current	12
`args` *glob*	Set the argument list	15

B.3 Substitution and Pattern Commands

Command	Effect	Ch.
`s/pat/rep/[flags]`	Substitute on current line	14
`%s/pat/rep/[flags]`	Substitute across whole buffer	14
`&`	Repeat last `:s`	14
`g/pat/cmd`	Run *cmd* on every matching line	15
`v/pat/cmd` (`:g!`)	Run *cmd* on every non-matching line	15

B.4 Execution and Automation

Command	Effect	Ch.
`normal` *keys*	Execute Normal-mode keystrokes literally	15
`execute` *expr*	Evaluate expression, run result as Ex command	15
`argdo` *cmd*	Run *cmd* on every file in argument list	15
`bufdo` *cmd*	Run *cmd* on every open buffer	15
`windo` *cmd*	Run *cmd* on every open window	15
`call` *Func*()	Call a Vimscript function	17
`let` *var* = *val*	Assign a variable or register	17
`if/elseif/else/endif`	Conditional block	17
`while/endwhile`	Loop block	17
`function!/endfunction`	Define a function	17
`autocmd` *event pattern cmd*	Bind *cmd* to an event	17
`augroup` *name*	Group related autocommands	17

B.5 Shell Interaction

Command	Effect	Ch.
`!`*cmd*	Run shell command, show output	18
`r !`*cmd*	Insert command output into buffer	18
`%!`*cmd*	Filter whole buffer through *cmd*	18
`w !`*cmd*	Send buffer to *cmd* as stdin	18
`cd` *dir*	Change working directory	18
`sh`	Start an interactive sub-shell	18

B.6 Build, Search, and Quickfix

Command	Effect	Ch.
`make`	Run build command, populate quickfix list	19
`grep` *pattern*	Search externally, populate quickfix list	19
`vimgrep` *pattern files*	Search with Vim's own regex engine	19
`copen`	Open quickfix window	19
`cnext` / `cprevious`	Next/previous quickfix entry	19

B.7 History and Inspection

Command	Effect	Ch.
`history` (`:his`)	Show command-line history	13
`jumps`	Show the jump list	11
`undolist`	Show undo-tree branch points	9
`undo` *n*	Jump to undo-tree state *n*	9
`digraphs`	Show the digraph table	8
`help` *topic* (`:h`)	Open help for *topic*	2

B.8 Options

Command	Effect	Ch.
`set` *option*	Enable a boolean option	3
`set no`*option*	Disable a boolean option	3
`set` *option=value*	Set a value option	3
`set all`	Print all current option values	E

Appendix C

Every Text Object

Every text object catalogued in Chapter 6, in inner (`i`) and around (`a`) form. Each is valid as the target of any operator from Appendix A's operator table, per the grammar of Chapter 4.

Inner	Around	Selects
`iw`	`aw`	Word (punctuation-sensitive); around includes adjacent whitespace
`iW`	`aW`	WORD (whitespace-delimited); around includes adjacent whitespace
`is`	`as`	Sentence; around includes trailing whitespace
`ip`	`ap`	Paragraph; around includes trailing blank line
`i"`	`a"`	Double-quoted string; around includes the quotes
`i'`	`a'`	Single-quoted string; around includes the quotes
`` i` ``	`` a` ``	Backtick-quoted string; around includes the quotes
`i(` / `ib`	`a(` / `ab`	Parenthesized region; around includes parentheses
`i{` / `iB`	`a{` / `aB`	Curly-braced region; around includes braces
`i[`	`a[`	Square-bracketed region; around includes brackets
`i<`	`a<`	Angle-bracketed region; around includes brackets
`it`	`at`	Content between a markup tag pair; around includes the tags

Nested delimiters (parentheses, braces, brackets, angle brackets) select the innermost enclosing pair by default; repeating the same text object while a Visual selection is active, per Chapter 6, expands the selection to the next enclosing pair. Quote text objects search only the current line, per the same chapter's discussion of why that restriction exists.

Appendix D

Every Register

Every register covered in Chapter 10, addressed with a leading `"` in Normal mode or `@` in Vimscript.

Register	Holds
`"` (unnamed)	Most recent deletion, change, or yank
`a`–`z`	Named registers; lowercase overwrites, uppercase (`A`–`Z`) appends
`0`	Most recent yank specifically (unaffected by subsequent deletions)
`1`–`9`	Automatically shifted history of line-or-larger deletions
`-` (small delete)	Most recent deletion smaller than one line
`:`	Most recently executed Ex command (read-only)
`/`	Most recent search pattern (read-only)
`.`	Most recently inserted text (read-only)
`%`	Current file name (read-only)
`#`	Alternate file name (read-only)
`=`	Expression register; evaluates a typed Vimscript expression
`*`	System selection/clipboard (platform-dependent)
`+`	System clipboard (X11 systems specifically)
`_`	Black hole register; discards on write, empty on read

Appendix E

Every Option

Every option named in this book, set with `:set` per Chapter 3's syntax. Boolean options are toggled with `set` *option* / `set no`*option*; value options are set with `set` *option*=*value*.

Option	Effect	Ch.
`nocompatible`	Disable classic-vi backward-compatibility behaviors	3
`syntax`	Enable syntax highlighting (`:syntax on`)	3
`number`	Display line numbers	3
`relativenumber`	Display line numbers relative to the cursor	E
`incsearch`	Show search matches while typing	3
`ignorecase`	Case-insensitive search by default	3
`smartcase`	Override `ignorecase` when pattern has uppercase	3
`hlsearch`	Highlight all search matches	E
`expandtab`	Insert spaces rather than literal tabs	3
`shiftwidth`	Width of one indent level	3
`softtabstop`	Width of a tab keystroke in Insert mode	3
`tabstop`	Display width of a literal tab character	E
`undofile`	Persist undo history to disk across sessions	9
`undodir`	Directory for persistent undo files	9
`textwidth`	Wrap width used by `gq`/`gw` and auto-wrap	7
`equalprg`	External program used by the = operator	7
`makeprg`	Build command run by `:make`	19
`iskeyword`	Characters considered part of a "word"	1
`spell`	Enable spell-checking	20
`wildmenu`	Visual, navigable command-line completion	13
`path`	Additional directories searched by `gf`/`:find`	E
`guioptions`	gVim interface elements (menu, toolbar) shown	3
`guifont`	gVim display font	3
`guicursor`	gVim cursor appearance and blink behavior	E
`shada` (formerly `viminfo`)	What session state persists across restarts	11

Appendix F

Every Built-in Function

The built-in Vimscript functions that appear in this book, each callable from the expression register (Chapter 10), the replacement-expression syntax of `:s` (Chapter 14), or ordinary Vimscript (Chapter 17). This is a small fraction of Vim's full built-in function library, documented completely under `:help functions`; only those used elsewhere in this book are listed here.

Function	Returns	Ch.
`line('.')`	Current line number	17
`line('$')`	Last line number of the buffer	17
`printf(fmt, ...)`	A formatted string, C-`printf`-style	17
`strftime(fmt)`	The current date/time, formatted per *fmt*	14
`@`*reg*	The contents of register *reg*, as an expression	14

Two syntactic contexts are worth distinguishing, since both are covered in this book and are easy to conflate: a bare function call such as `printf(...)` is valid directly inside `:execute`'s expression argument or inside a Vimscript `:let` assignment, per Chapter 17; the same call, used inside a `:s` replacement, additionally requires the leading `\=` marker from Chapter 14 to signal that the replacement text should be evaluated rather than inserted literally.

Appendix G

Regular Expression Reference

The regex constructs used across this book's substitution and pattern-matching examples (Chapters 5, 14, and 15), given in Vim's default (`magic`) mode unless noted. Prefixing a pattern with `\v` switches to very-magic mode, in which the unescaped forms in the right-hand column below apply directly.

G.1 Anchors and Classes

Default	Very Magic	Matches
`^`	`^`	Beginning of line
`$`	`$`	End of line
`.`	`.`	Any single character
`\<`	`<`	Beginning of word
`\>`	`>`	End of word
`[string]`	`[string]`	Any single character in *string*
`[^string]`	`[^string]`	Any character not in *string*
`[a-p]`	`[a-p]`	Any character in range a–p
`\d`	`\d`	Any digit
`\w`	`\w`	Any word character
`\s`	`\s`	Any whitespace character

G.2 Quantifiers and Grouping

Default	Very Magic	Matches
`*`	`*`	Zero or more of the preceding atom
`\+`	`+`	One or more of the preceding atom
`\?`	`?`	Zero or one of the preceding atom
`\(...\)`	`(...)`	Capture group
`|`	`|`	Alternation
`\zs`	`\zs`	Reset reported match start (zero-width)
`\`	`\`	Escape the next character's special meaning

G.3 Replacement Syntax

Construct	Effect in a `:s` replacement
`\1`, `\2`, ...	Text captured by the corresponding group
`&` (or `\0`)	The entire matched text
`\u`	Capitalize the next character
`\l`	Lowercase the next character
`\U` ... `\E`	Uppercase everything until `\E`
`\L` ... `\E`	Lowercase everything until `\E`
`\r`	A literal newline (distinct from `\n` in the pattern)
`\=`*expr*	Evaluate *expr* as Vimscript; insert the result

Chapter 14 works through several complete patterns built from these pieces, including sentence capitalization using `\zs` and a very-magic capture-group example; those worked examples are the more useful starting point for constructing a new pattern than this table alone, which is meant for looking up a single construct's syntax once its role in the pattern is already understood.

Appendix H

Command-Line Arguments

Arguments to the `vim` command itself, invoked from the shell, as covered across Chapters 3, 9, 12, and 15.

Argument	Effect	Ch.
`vim` *file*	Edit *file*	3
`vim` *file1 file2 ...*	Edit multiple files (populates the argument list)	15
`-c` *cmd*	Run Ex command *cmd* after loading, before display	15
`-r` *file*	Recover *file* from its swap file after a crash	9
`-R`	Open in read-only mode	H
`-d` *file1 file2*	Open in diff mode (equivalent to `vimdiff`)	19
`-S` *file*	Restore a saved session	12
`--version`	Print version and compiled-in feature list	3

Multiple `-c` arguments are executed in the order given, each a complete Ex command exactly as it would be typed after `:` in a running session, which is what makes `vim -c '%s/old/new/g' -c 'wq' file.txt` a complete, non-interactive batch edit: the substitution runs, then the write-and-quit, with no editor session ever visibly opened.

Appendix I

vimrc Cookbook

Every vimrc line introduced across this book, collected into one place. This is not a single recommended configuration to copy wholesale; it is the complete set of individual settings this book has justified individually, in the chapter noted, for a reader to select from according to their own needs, following the incremental spirit of Chapter 3's original minimal vimrc.

```
" Core sanity (Chapter 3)
set nocompatible
syntax on
set number
set incsearch
set ignorecase
set smartcase

" Indentation (Chapter 3)
set expandtab
set shiftwidth=4
set softtabstop=4

" Undo persistence across sessions (Chapter 9)
set undofile
" set undodir=~/.vim/undodir " optional: centralize undo files

" Command-line completion (Chapter 13)
set wildmenu

" gVim interface, if applicable (Chapter 3)
set guioptions-=m " remove menu bar
set guioptions-=T " remove toolbar

" Prose and writing (Chapter 20)
" set spell " enable per-file with :set spell instead
" set textwidth=80 " set per-filetype rather than globally

" A custom mapping (Chapter 17)
nnoremap <C-s> <C-a>h

" An autocommand (Chapter 17)
augroup PythonIndent
    autocmd!
```

```
    autocmd BufWritePre *.py normal! gg=G
augroup END
```

The two options left commented out, `spell` and `textwidth`, are shown that way deliberately: both are more useful set per-file-type or invoked on demand than enabled globally for every buffer regardless of content, and a reader adapting this cookbook should treat every line here as a starting point for judgment rather than a fixed prescription, consistent with this book's general preference throughout for understanding a setting's effect over copying it uncritically.

Appendix J

Historical Evolution of Vim

A compact timeline complementing the fuller historical discussion in Chapter 1.

Era	Development
Early 1970s	`ed`, a line editor, establishes the command language later inherited by `ex` and, through it, Vim's own Ex mode: line addressing, `:s`, `:g`, and range syntax all trace to this era.
Late 1970s	Bill Joy writes `vi`, adding a full-screen, character-addressable visual mode atop `ex`'s existing line-oriented command language, establishing the dual nature (visual editing plus an underlying line-oriented Ex language) that Vim still exhibits.
1988	Bram Moolenaar begins Vim ("Vi IMitation," later reinterpreted as "Vi IMproved") for the Amiga, as a vi-compatible clone.
1990s onward	Vim adds multi-level undo, syntax highlighting, split windows, Vimscript, and a large ecosystem of built-in and pluggable extensions, while preserving vi's command grammar essentially intact.
2014	Neovim forks from Vim, preserving the same operator-motion-text-object grammar while modernizing the implementation: a different plugin architecture, first-class Lua scripting alongside Vimscript, and a design oriented toward embedding the editor in other tools.

The through-line worth emphasizing, restated from Chapter 1: the operator-motion grammar covered throughout Part II, the register model of Chapter 10, and the Ex language of Part V have remained substantially stable across every era in this table. A command learned from this book will continue to work, unmodified, regardless of which specific era's implementation a reader happens to be running.