

COURSE

Learning Vim

9 Chapters

Philip Osztromok

Generated with Claude

LEARNING VIM

Contents

Chapter 1	Introduction	2
Chapter 2	Basics	10
Chapter 3	Moving Around	20
Chapter 4	Changing Text	32
Chapter 5	Marks	45
Chapter 6	Buffers, Windows, and Tabs	54
Chapter 7	Configuration	64
Chapter 8	Day to Day Vim	77
Chapter 9	Appendix A — Going Further	90

Learning Vim

Chapter 1 — Introduction

Why Vim?

Vim is a text editor unlike any other. Most editors behave the way you expect — you open a file, click where you want to type, and start editing. Vim does not work that way, and that difference is exactly what makes it powerful. Once you have learned Vim's way of thinking, you can edit text faster, more precisely, and with less hand movement than in almost any other editor.

Here is what makes Vim worth the investment:

- **It is everywhere.** Vim — or its predecessor `vi` — comes pre-installed on virtually every Unix-based system. Linux servers, macOS, embedded devices, cloud VMs: if you can SSH into it, Vim is almost certainly available. You will never be stranded without an editor.
- **It runs in a terminal.** You do not need a graphical interface. This is critical when working on remote servers, in containers, or in recovery environments where a GUI simply does not exist.
- **It is fast.** Opening, editing, and closing a file in Vim takes seconds. For quick configuration file edits — the kind of task you do dozens of times a day as a developer or sysadmin — this adds up to a significant amount of saved time.
- **It is composable.** Vim commands combine like building blocks. Once you understand the grammar — *operator* + *motion* — you can do things you have never explicitly learned just by combining parts you already know.
- **It is enduring.** Vim was released in 1991, built on the ideas of `vi` from 1976. After more than 30 years it is still actively developed and widely used. This kind of longevity (sometimes called the *Lindy effect* — the longer something has survived, the longer it is likely to continue surviving) is a strong signal that the time you invest in learning it will pay off for the rest of your career.

> **The Stack Overflow statistic:** According to Stack Overflow's own data, the question "How do I exit Vim?" has been viewed millions of times, with around 80 people per hour looking it up at its peak. This is not evidence that Vim is bad — it is evidence that Vim is

> surprising. The moment you understand *why* exiting Vim is non-obvious (modes, which we cover in Chapter 2), that surprise transforms into clarity.

The Learning Curve — An Honest Picture

Vim has a well-earned reputation for a steep learning curve. A common joke in developer circles is that Vim's learning curve is not just steep but *vertical* — you put a lot of effort in before you get any results out. That is partially true, but it is more nuanced than the joke suggests.

A better analogy is learning to drive a manual (stick-shift) car:

- The first time you sit behind the wheel, every action is conscious and effortful. You have to think about the clutch, the gear, the mirror, the steering — all at once.
- You will stall. You will forget which gear you are in. The car will do things you did not expect.
- After a few weeks of practice, the conscious effort fades. You change gear without thinking about it. Your hands know what to do before your brain has finished forming the instruction.
- At that point, you are more in control of the car than you ever were when it was automatic — because you understand what it is doing and why.

Vim follows the same arc. The awkward phase is real but it is finite. Most people find that after one to two weeks of daily use (even just 20–30 minutes a day), the basic movements and editing commands start to feel natural. After a month, they start to miss Vim when they are forced to use something else.

The trick is to **not abandon Vim during the awkward phase**. It helps to have a project or file you actually need to edit — real motivation to push through the discomfort. This course is designed to get you through that phase as efficiently as possible by focusing on the commands you will use every day, explained in a way that makes the underlying logic visible.

Vim's Modal Design — A First Look

The most important thing to understand about Vim before you even open it is that it is a **modal editor**. This is the idea behind all of Vim's apparent strangeness, and once you understand it everything else clicks into place.

Most editors have one mode: you are always typing text. Vim has several modes, and the meaning of every key you press depends on which mode you are currently in. The three you will use most are:

Mode	What it is for	How to enter it
`NORMAL`	Navigating, deleting, copying, and issuing commands. This is Vim's *home base* — the mode you should return to when you are not actively inserting text.	Press `Esc` from any other mode
`INSERT`	Typing text into the file, just like a conventional editor.	Press `i` in Normal mode
`COMMAND`	Running editor commands — saving, quitting, searching, changing settings.	Press `:` in Normal mode

The reason most people get stuck when they first open Vim is that they are dropped straight into Normal mode. They start typing, and nothing appears (or unexpected things happen) because keystrokes in Normal mode are commands, not text. This single piece of knowledge — **Vim starts in Normal mode** — eliminates the most common source of confusion.

We will explore each mode thoroughly in the chapters ahead. For now, just hold onto this mental model: Normal mode is your home; Insert mode is where you type; Command mode is where you talk to the editor.

Installation

Checking whether Vim is already installed

On macOS and most Linux distributions, Vim is pre-installed. Open a terminal and run:

```
vim --version
```

If Vim is installed you will see several lines of output beginning with the version number:

```
VIM - Vi IMproved 9.0 (2022 Jun 28, compiled ...)
Included patches: 1-749
...
```

The first line tells you the version. Vim 8 or 9 is recommended — both support all the features in this course. If you see `command not found`, follow the installation steps below.

> **vi vs vim:** Many systems also have `vi` — the original editor that Vim is based on. Running `vi --version` on such a system often actually invokes Vim in compatibility mode. For this course, always use `vim` explicitly to ensure you have the full feature set.

macOS

macOS ships with Vim, but it may be an older version. To install the latest:

```
# Using Homebrew (recommended)
brew install vim

# Verify the new version is on your PATH
which vim     # should show /usr/local/bin/vim or /opt/homebrew/bin/vim
vim --version
```

Linux

Use your distribution's package manager:

```
# Debian / Ubuntu
sudo apt update && sudo apt install vim

# Fedora / RHEL / CentOS
sudo dnf install vim

# Arch Linux
sudo pacman -S vim

# Alpine (common in Docker containers)
apk add vim
```

Windows

Vim is not pre-installed on Windows. Download the official installer from **www.vim.org**:

1. Go to **vim.org → Download → PC, MS-DOS and MS-Windows**
2. Download the latest **gvim_X.Y.exe** installer (this installs both the terminal `vim` and the GUI `gvim`)
3. Run the installer with default options
4. Open a new Command Prompt or PowerShell window and run `vim --version` to verify

> **Windows Subsystem for Linux (WSL):** If you are a developer on Windows, running Vim inside WSL (Ubuntu or similar) gives you the full Linux experience and is the recommended approach for this course. Install WSL first (`wsl --install` in an elevated PowerShell), then install Vim inside your Linux distribution as described above.

Opening Vim & the Interface

Starting Vim

To open an existing file:

```
vim filename.txt
```

To create a new file:

```
vim newfile.txt
```

To open Vim with no file (you can name the file when saving):

```
vim
```

Reading the interface

When Vim opens a new empty file you will see something like this:

```
~
~
~
~
~
~
                                                0,0-1         All
```

Each `~` represents a line that does not exist — it is Vim telling you "there is no content here". The status bar at the bottom right shows your position in the file: `0,0-1` means line 0 (no lines yet), column 0. `All` means the entire file is visible on screen.

When you open an existing file the content fills the screen and the status bar shows the filename and line/column position:

```
This is the first line of my file
This is the second line
~
~
"myfile.txt" 2 lines, 58 characters
```

Enabling line numbers

Line numbers make navigation much easier — you can jump directly to a specific line, and error messages from compilers and interpreters almost always refer to line numbers. To enable them, switch to Command mode and run:

```
:set number
```

Type the colon (`:`) first — this moves you from Normal mode into Command mode and you will see the cursor jump to the bottom of the screen. Then type `set number` and press `Enter`. Line numbers will appear on the left of every line.

```
  1 This is the first line of my file
  2 This is the second line
```

```
  3 ~
  4 ~
```

To turn line numbers off:

```
:set nonumber
```

This setting only lasts for the current session. In Chapter 8 we will cover the `.vimrc` configuration file, where you can make it permanent with a single line.

> **Relative line numbers:** Vim also supports *relative* line numbers, which show the distance from the current line rather than absolute line numbers. Many experienced Vim users prefer these because they make vertical motion commands (covered in Chapter 3) more intuitive. Enable them with `:set relativenumber`.

Your First Vim Session — A Safe Walkthrough

Before we go any further, here is a simple sequence you can follow right now to open Vim, make a small edit, and get out safely. Think of it as a first driving lesson in a car park.

```
# 1. Open a new test file
vim test.txt
```

Vim opens in `NORMAL` mode. The screen is empty (or shows the file contents). Nothing you type will appear as text yet.

1. Press `i` — this enters `INSERT` mode. You will see `-- INSERT --` at the bottom of the screen.
2. Type a few words: `Hello from Vim!`
3. Press `Esc` — this returns you to `NORMAL` mode. The `-- INSERT --` indicator disappears.
4. Type `:w` and press `Enter` — this saves (writes) the file.
5. Type `:q` and press `Enter` — this quits Vim.

Congratulations — you have successfully entered, edited, saved, and exited Vim. These five steps cover the most common stumbling block new users face.

Made a mistake and want to quit without saving? Use `:q!` — the exclamation mark forces Vim to quit and discard any unsaved changes. This is your escape hatch whenever something has gone wrong and you just want out.

Commands Introduced in This Chapter

CHAPTER 1 — COMMAND REFERENCE

`vim --version`	Check whether Vim is installed and display the version number
`vim <filename>`	Open a file in Vim (creates it if it does not exist)
`i`	Enter Insert mode at the cursor position NORMAL → INSERT
`Esc`	Return to Normal mode from any other mode
`:set number`	Display line numbers
`:set nonumber`	Hide line numbers
`:set relativenumber`	Display line numbers relative to the current line
`:w`	Write (save) the current file
`:q`	Quit Vim (only works if there are no unsaved changes)
`:q!`	Force quit — discard unsaved changes and exit
`:wq`	Write and quit in one step

Learning Vim

Chapter 2 — Basics

Understanding the Interface

When you open Vim you are met with a deliberately minimal screen. There are no toolbars, no menus, no panels. This is intentional — every pixel is reserved for your file. Let us decode the few things that *are* on screen.

The tilde lines

Empty lines at the bottom of the editing area are shown as a single `~` (tilde) character. These are not part of your file — they are Vim's way of saying "there is no content here". This distinction matters: a blank line in your file (a line that exists but contains nothing) looks different from a `~` line. A blank line is truly empty; a `~` line does not exist at all.

```
1 This is line one of my file
2 This is line two
3
4 ~
5 ~
6 ~
```

In the example above, line 3 is a real blank line (you pressed Enter to create it). Lines 4–6 with tildes are beyond the end of the file.

The status line

The bottom of the Vim window is the status area. It serves multiple purposes depending on context:

- **Mode indicator:** When you enter Insert, Visual, or Replace mode, the current mode name appears here (e.g. `-- INSERT --`).
- **Command input:** When you press `:`, your typed command appears here.

- **File information:** After opening a file, Vim shows its name, line count, and character count briefly.
- **Position:** The bottom-right corner shows your cursor position as `line,column`.

```
  1 Hello, Vim!
  2 ~
  3 ~
"hello.txt" [New File]                    1,1           All
```

Reading the status line above: the file is called `hello.txt`, it is new (not yet saved to disk), the cursor is at line 1, column 1, and `All` means the entire file fits on screen.

Vim Modes — A Complete Picture

In Chapter 1 we introduced three modes. Now let us look at all five you will encounter, understand how they relate to each other, and learn the correct way to move between them.

Mode	Indicator	Purpose	Enter from Normal
NORMAL	*(no indicator)*	Navigation, text manipulation, issuing commands. Vim's home base.	It is the default — press `Esc` to return here from anywhere.
INSERT	`-- INSERT --`	Typing text into the file, just like a conventional editor.	`i`, `a`, `o`, `I`, `A`, `O`, `s`, `c`
VISUAL	`-- VISUAL --`	Selecting blocks of text to copy, delete, or transform.	`v` (character), `V` (line), `Ctrl+v` (block)
COMMAND	`:` prompt at bottom	Running editor commands — save, quit, search/replace, settings.	`:`
REPLACE	`-- REPLACE --`	Overwriting existing text character by character (like Insert with overwrite).	`R`

Visual mode and Replace mode are covered in depth in later chapters. For now the key insight is this: **Normal mode is the hub.** Every other mode is reached from Normal mode, and pressing `Esc` always brings you back to Normal mode.

```
Mode transition diagram:

  INSERT  ←── i, a, o ──┐
  VISUAL  ←── v, V ─────┤
  COMMAND ←── : ────────┼──── NORMAL  (always return with Esc)
  REPLACE ←── R ────────┘
```

Normal mode in depth

Normal mode is not a "waiting" mode — it is an active editing mode in which every key is a command. This is Vim's core philosophy: you spend most of your time navigating and manipulating text, with only brief excursions into Insert mode to type new content. In a typical editing session you might be in Normal mode 70–80% of the time.

A few examples of what single keys do in Normal mode (we cover all of these in later chapters):

Key	Action in Normal mode
`d`	Delete operator (combine with a motion: `dw` = delete word)
`y`	Yank (copy) operator
`p`	Put (paste) after cursor
`u`	Undo
`D`	Delete from cursor to end of line
`G`	Jump to the last line of the file
`x`	Delete the character under the cursor

> **The most important rule in Vim:** If something unexpected is happening — keys are not doing what you expect, text is being deleted when you want to type, or you are lost —

> press `Esc` repeatedly until the mode indicator disappears from the status line. You are back in Normal mode and nothing more will change unexpectedly.

Returning to Normal mode: Esc vs Ctrl+C

There are two ways to return to Normal mode from Insert or Visual mode:

- `Esc` — the standard way. Always works, always safe.
- `Ctrl+C` — an alternative that most users find slightly faster to reach. It works identically to `Esc` in almost all situations, with one subtle difference: it will not trigger the `InsertLeave` autocommand event (relevant only if you use advanced plugins or autocommands — ignore this for now).

Many experienced Vim users remap `Caps Lock` to `Esc` at the operating system level, since `Esc` is used so frequently and is awkward to reach. Another popular choice is to map `jj` or `jk` (typed quickly) to `Esc` in the Vim config. We will cover remapping in Chapter 8.

Insert mode — the creative phase

Insert mode is where you write. Once you are in Insert mode, Vim behaves much like any other text editor: the arrow keys move the cursor, `Backspace` deletes the previous character, `Enter` creates a new line, and what you type appears on screen.

The important mental discipline is this: **when you have finished writing a section of text, press `Esc` immediately.** Do not stay in Insert mode while you read what you have written or think about what to write next. Return to Normal mode. This habit is what separates Vim users who feel fast from those who feel slow.

> **Insert mode keyboard shortcuts:** Even while in Insert mode there are a handful of useful shortcuts that do not require switching to Normal mode first:
>
> - `Ctrl+W` — delete the word before the cursor (equivalent to `Backspace` × word)
> - `Ctrl+U` — delete from the cursor back to the start of the line
> - `Ctrl+T` — indent the current line one shiftwidth to the right
> - `Ctrl+D` — remove one level of indentation from the current line

File Operations

Opening files

You have already seen `vim filename` from the shell. There are several variations worth knowing:

```
# Open a single file
vim notes.txt

# Open multiple files (cycles through them with :n and :prev)
vim file1.txt file2.txt file3.txt

# Open a file and jump to a specific line number
vim +42 script.sh

# Open a file and jump to the first occurrence of a pattern
vim +/"search term" document.txt

# Open a file in read-only mode (prevents accidental edits)
vim -R important.conf
```

Opening a different file from within Vim

You do not need to quit Vim to open another file. The `:e` command (short for *edit*) loads a new file into the current buffer:

```
:e filename.txt
```

If you have unsaved changes in the current file, Vim will warn you:

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

Either save first with `:w`, or force the switch and discard changes with `:e!`.

Tab completion is available when typing filenames after `:e`. Start typing the filename and press `Tab` to autocomplete. If there are multiple matches, press `Tab` again to cycle through

them, or press `Ctrl+D` to display all matches in a list below the command line.

```
# Example: typing :e kno<Tab> might complete to :e knowledge.txt
:e kno<Tab>       →  :e knowledge.txt
:e script<Tab>    →  :e script_   (cycles: script_one.sh, script_two.sh ...)
```

Saving files

The write command `:w` has several useful forms:

```
# Save the current file (overwrite)
:w

# Save to a different filename (original file unchanged — like "Save As")
:w newfilename.txt

# Save a specific range of lines to a new file (lines 5 through 12)
:5,12w excerpt.txt

# Append the current file's contents to an existing file
:w >> log.txt

# Force save a file opened read-only (if you have filesystem permissions)
:w!
```

> **Save vs Save As:** `:w newname.txt` writes the current buffer's contents to `newname.txt`, but Vim continues to think you are editing the *original* file. To rename the file you are working on (i.e. change which file this buffer is associated with), use `:saveas newname.txt` instead — this writes the new file and changes the buffer's name.

Checking the current file

At any point in Normal mode, pressing `Ctrl+G` displays a summary of the current file in the status line:

```
"knowledge.txt" line 3 of 10 --30%-- col 1
```

This tells you the filename, your current line number, the total number of lines, what percentage through the file you are, and your column position. It is a fast way to reorient yourself when you have been navigating through a long file.

For more detail, the command `:f` (or `:file`) shows the same information.

Quitting Vim

There are more quit variants than you might expect, and knowing all of them prevents frustration:

Command	Behaviour
`:q`	Quit. Only works if there are no unsaved changes.
`:q!`	Force quit. Discards all unsaved changes.
`:wq`	Write and quit. Saves, then exits.
`:x`	Write and quit — but only writes if changes were made. Slightly smarter than `:wq`.
`ZZ`	Normal mode shortcut for `:x` — no colon needed.
`ZQ`	Normal mode shortcut for `:q!` — force quit without saving.
`:qa`	Quit all open buffers/windows.
`:qa!`	Force quit everything — all buffers, discarding any unsaved changes.

Anatomy of `:qa!`: Breaking this command down makes every Vim command easier to learn:

- `:` — enter Command mode
- `q` — the quit command
- `a` — modifier meaning "all" (apply to all open buffers)
- `!` — modifier meaning "force" (do not ask for confirmation)

Vim commands often follow this pattern of a base command plus optional modifiers. Once you learn the modifiers they apply consistently across many commands.

Practical Walkthrough — Building a Knowledge File

Let us put everything from this chapter together in a practical session. We will create a file, add some content, practise moving between modes, and explore several ways to save and quit. Follow along in your own terminal.

Workflow — Creating knowledge.txt

1	`vim knowledge.txt`	Open (create) the file from your terminal. Vim opens in Normal mode — the status line shows `"knowledge.txt" [New File]`.
2	`:set number`	Enable line numbers so you can follow along easily. Press Enter after typing the command.
3	`i`	Enter Insert mode. The status line shows `-- INSERT --`. You are now ready to type.
4	`(type your content)`	Type several lines, for example: *"The only source of knowledge is experience" — Albert Einstein*. Press Enter between lines.
5	`Esc`	Return to Normal mode. The `-- INSERT --` indicator disappears. Good habit: return here whenever you pause to read.
6	`Ctrl+G`	Check your position. The status line shows the filename, current line, total lines, and column. Useful for orientation.
7	`:w`	Save the file. The status line confirms: `"knowledge.txt" written`.
8	`:w backup.txt`	Save a copy to `backup.txt`. Note that you are still editing `knowledge.txt`.
9	`:e backup.txt`	Switch to the backup file to verify the copy is correct. Tab completion works here — try typing `:e bac<Tab>`.
10	`:e knowledge.txt`	Switch back to the original file.

11	`ZZ`	Save and quit in one keystroke (Normal mode shortcut — no colon needed). You are back at the shell prompt.

Experimenting with the quit variants

Reopen the file and practice the different quit commands so they become muscle memory:

```
vim knowledge.txt

# Make a change (enter Insert mode, type something, press Esc)
# Then try quitting without saving:
:q          # Vim refuses — there are unsaved changes
             # Output: E37: No write since last change (add ! to override)

:q!         # Force quit — change is discarded

# Reopen, make another change, then save and quit:
vim knowledge.txt
# ... make a change ...
ZZ           # save and quit — fastest method
```

Commands Introduced in This Chapter

CHAPTER 2 — COMMAND REFERENCE

`Ctrl+C`	Alternative to `Esc` — return to Normal mode
`Ctrl+G`	Display filename, line number, and position in the status line `NORMAL`
`ZZ`	Write and quit (only writes if changes were made) `NORMAL`
`ZQ`	Force quit without saving `NORMAL`
`Ctrl+W (Insert)`	Delete the word before the cursor `INSERT`
`Ctrl+U (Insert)`	Delete from cursor to start of line `INSERT`
`:e <file>`	Open (edit) a file; Tab key completes the filename
`:e!`	Re-load the current file from disk, discarding unsaved changes
`:w <file>`	Write buffer contents to a new filename (Save As)

`:saveas <file>`	Write to a new filename and switch the buffer to that file
`:x`	Write (only if changed) and quit — smarter version of `:wq`
`:qa`	Quit all open buffers
`:qa!`	Force quit all buffers, discarding all unsaved changes
`vim +N <file>`	Open file and jump to line N from the shell
`vim -R <file>`	Open file in read-only mode from the shell

Learning Vim

Chapter 3 — Moving Around

Why Movement Matters

In most editors, moving around the file is an afterthought — you use the mouse, or hold arrow keys until you get where you want to go. In Vim, movement is a first-class skill. Because all editing happens in Normal mode using keyboard commands, the speed at which you can navigate directly determines the speed at which you can edit.

Vim's movement commands are also the foundation of its *operator + motion* grammar. Commands like "delete the next three words" or "copy from here to the end of the paragraph" are built by combining an editing operator with a movement command. Learning to move precisely and quickly therefore unlocks not just navigation but the entire editing system.

Everything in this chapter is a `NORMAL` mode command unless stated otherwise.

The HJKL Keys — Character Movement

Vim uses `h`, `j`, `k`, and `l` as arrow keys. This is not an arbitrary choice — it has its roots in the original ADM-3A terminal from the 1970s, on which the arrow symbols were actually printed on those keys. The layout has a practical advantage: these four keys sit on the home row of the keyboard, right where your fingers already rest. You never need to move your hand to the arrow cluster.

Key	Direction	Memory aid
`h`	Left (←)	Leftmost of the four keys
`j`	Down (↓)	The letter j has a tail that points down
`k`	Up (↑)	k is above j on the keyboard
`l`	Right (→)	Rightmost of the four keys

> **Arrow keys still work** — Vim accepts arrow keys for movement in both Normal and Insert mode. However, using them forces your right hand off the home row and across to the arrow cluster on every navigation. Most Vim users find that after a few days of consciously using HJKL, it becomes automatic and they never want to go back.

Multiplying movement with counts

Any movement command can be prefixed with a number to repeat it that many times. This is one of the most efficient features of Vim navigation — instead of holding a key down and watching the cursor drift, you specify exactly how far you want to go.

```
5j      # move 5 lines down
10l     # move 10 characters to the right
3k      # move 3 lines up
8h      # move 8 characters to the left
```

Counts work with almost every movement command in this chapter — not just HJKL. When you see a new command, assume a count prefix will work unless stated otherwise.

> **Relative line numbers and counts:** When `:set relativenumber` is enabled (introduced in Chapter 1), the number shown next to each line is its distance from the cursor. This makes count-based jumps trivial — you glance at the line you want, read off its number, and type that number followed by `j` or `k`.

Word-Level Movement

Moving character by character is slow even with counts. For navigating within a line or across a paragraph, *word-level* movements are far more efficient.

Key	Movement
`w`	Forward to the **start** of the next word
`W`	Forward to the start of the next WORD (whitespace-delimited — ignores punctuation)

`e`	Forward to the **end** of the current (or next) word
`E`	Forward to the end of the current WORD
`b`	Backward to the **start** of the current (or previous) word
`B`	Backward to the start of the current WORD
`ge`	Backward to the **end** of the previous word

> **word vs WORD:** In Vim, a lowercase *word* is a sequence of letters, digits, and underscores — roughly what a programmer would call an identifier. An uppercase *WORD* is any sequence of non-whitespace characters. So in the text `file-name.txt`, `w` sees three words (`file`, `-`, `name.txt` approximately), whereas `W` sees one WORD. Use lowercase for precision within punctuation-heavy text; use uppercase for coarser, faster jumps.

Example — word movement on a line

```
The quick brown fox jumps over the lazy dog
```

Starting with cursor on "T" in "The":

Command	Cursor lands on
`w`	q (start of "quick")
`3w`	f (start of "fox")
`e`	e (end of "The")
`2e`	k (end of "quick"… wait, end of "brown")
`b`	T (back to start of "The")
`4w`	j (start of "jumps")

Line Movement

Moving within a line precisely is something you do constantly. These commands get you to the right position instantly:

Key	Movement
`0`	Jump to the **very beginning** of the line (column 1)
`^`	Jump to the **first non-whitespace** character of the line
`$`	Jump to the **end** of the line
`g_`	Jump to the **last non-whitespace** character of the line
`f{char}`	Find — jump **forward** to the next occurrence of *char* on the line
`F{char}`	Find — jump **backward** to the previous occurrence of *char* on the line
`t{char}`	Till — jump forward to just **before** *char* on the line
`T{char}`	Till — jump backward to just **after** *char* on the line
`;`	Repeat the last `f`, `F`, `t`, or `T` in the same direction
`,`	Repeat the last `f`, `F`, `t`, or `T` in the opposite direction

Example — f and t in action

```
background-color: rgba(255, 128, 0, 0.5);
```

Starting with cursor at the beginning of the line:

- `f(` — jumps to the `(` before `255`
- `t,` — jumps to just before the first `,` (cursor sits on `5`)
- `;` — repeats: jumps to just before the next `,`
- `f;` — jumps to the `;` at the end of the line

This is much faster than pressing `l` repeatedly or using the arrow keys to navigate within a long line.

The difference between `0` and `^`: `0` always goes to column 1, even if the line is indented. `^` goes to the first actual character, skipping any leading whitespace. In code,

you almost always want `^` — it puts you at the start of the statement, not at the indentation.

Sentence and Paragraph Movement

When working with prose or structured documents, jumping sentence-by-sentence or paragraph-by-paragraph is faster than any word-level navigation.

Key	Movement
`)`	Forward to the beginning of the next sentence
`(`	Backward to the beginning of the current (or previous) sentence
`}`	Forward to the beginning of the next paragraph (next blank line)
`{`	Backward to the beginning of the previous paragraph

What counts as a sentence? Vim defines a sentence as ending with `.`, `!`, or `?` followed by either end-of-line or at least two spaces. A paragraph is defined as a block of non-empty lines — it ends and begins at a blank line. In code, `}` and `{` are most useful for jumping between functions or blocks separated by blank lines.

As with all movement commands, counts work here too:

```
3}     # jump 3 paragraphs forward
2{     # jump 2 paragraphs backward
5)     # jump 5 sentences forward
```

File-Level Navigation

For large files you need to jump across long distances — to the top, the bottom, a specific line, or a screenful at a time.

Top and bottom

Key	Movement
`gg`	Jump to the **first line** of the file
`G`	Jump to the **last line** of the file
`{n}G`	Jump to **line n** (e.g. `42G` jumps to line 42)
`:{n}`	Jump to **line n** via the command line (e.g. `:302`)
`{n}%`	Jump to **n%** through the file (e.g. `50%` jumps to the midpoint)

> **`{n}G` vs `:{n}`:** Both jump to line *n*. `{n}G` is faster to type in Normal mode (just the number then `G`). `:{n}` is useful when you know the number from an error message — you can type it quickly after pressing `:`. Both are worth knowing.

Page scrolling

Key	Movement
`Ctrl+F`	Scroll **forward** one full page (keeps a couple of lines for context)
`Ctrl+B`	Scroll **backward** one full page
`Ctrl+D`	Scroll **down** half a page
`Ctrl+U`	Scroll **up** half a page
`Ctrl+E`	Scroll the screen down one line *without moving the cursor*
`Ctrl+Y`	Scroll the screen up one line *without moving the cursor*

Cursor positioning on screen

These three commands reposition the *view* around your cursor, not the cursor itself — useful for reading context around your current position:

Key	Action
`zz`	Scroll so the current line is in the **centre** of the screen

`zt`	Scroll so the current line is at the **top** of the screen
`zb`	Scroll so the current line is at the **bottom** of the screen

> `zz` **is one of the most underrated commands in Vim.** When you have just jumped to a line and want to read the surrounding context, `zz` centres it on screen instantly. It is also useful while reading — if the line you care about has drifted to the top or bottom of the screen, `zz` brings it back to eye level.

Searching

Search is arguably the fastest navigation tool in Vim for reaching a specific location. Instead of scrolling to where you think the text is, you tell Vim exactly what you are looking for and it takes you there instantly.

Forward and backward search

```
/pattern    # search forward for pattern — press Enter to jump
?pattern    # search backward for pattern — press Enter to jump
```

After pressing Enter, the cursor jumps to the first match. The matched text is highlighted across the entire file (if `:set hlsearch` is enabled). To navigate between matches:

Key	Action
`n`	Jump to the **next** match (in the direction of the original search)
`N`	Jump to the **previous** match (opposite direction)

The search wraps around — when you reach the end of the file Vim loops back to the beginning (you will see a message: `search hit BOTTOM, continuing at TOP`).

> **Clearing search highlights:** After a search, the highlighted matches stay on screen until you search for something else. To clear the highlights without searching, run

`:nohlsearch` (or its abbreviation `:noh`). Many users map this to a key in their config — for example `:nnoremap <Esc><Esc> :nohlsearch<CR>`.

Searching for the word under the cursor

Often the word you want to search for is already under your cursor. Rather than typing it out, use:

Key	Action
`*`	Search **forward** for the exact word under the cursor (whole-word match)
`#`	Search **backward** for the exact word under the cursor
`g*`	Search forward for the word under the cursor (partial match — finds it inside longer words too)
`g#`	Search backward (partial match)

`*` is particularly useful when reading code — place your cursor on a variable name and press `*` to jump through all its usages in the file.

Regular expressions in search

Vim's search supports regular expressions, making it enormously powerful for finding patterns rather than literal strings:

```
/ch.mb          # . matches any single character: finds "chamber", "chimb", "chombs
/^function      # ^ = start of line: finds lines beginning with "function"
/error$         # $ = end of line: finds lines ending with "error"
/\d\+           # \d = digit, \+ = one or more: finds any number
/colou\?r       # \? = zero or one: finds both "color" and "colour"
/\<word\>       # \< and \> = word boundaries: whole-word match (like * does automa
```

Case sensitivity: By default, Vim's search is case-sensitive. You can change this with `:set ignorecase`. A smarter option is `:set smartcase` — when enabled, searches are

> case-insensitive unless your pattern contains an uppercase letter, in which case they become case-sensitive. This is the behaviour most users prefer.

Search settings worth knowing

```
:set hlsearch      # highlight all matches in the file
:set incsearch     # move cursor as you type the pattern (incremental search)
:set ignorecase    # case-insensitive search
:set smartcase     # case-sensitive only when pattern has uppercase
:noh               # clear current search highlights
```

Putting It All Together — Practical Examples

Navigation commands become powerful when combined. Here are the worked examples from the course, showing how you think through a multi-step navigation problem.

Navigate to the 4th word of the 2nd-to-last sentence of the 3rd paragraph

```
3}     # jump 3 paragraphs forward (land at the start of paragraph 3)
2{     # oops — went too far? Step back 2 paragraphs
       # (adjust the count based on where you land)
(     # move back one sentence to the 2nd-to-last
3w     # move forward 3 words — now on the 4th word
```

The key insight is that you do not need to be perfect on the first jump. Vim navigation is iterative — overshoot, correct, refine.

Find the word immediately before "England"

```
/England    # jump to "England"
b           # move back one word — the word before "England"
```

Find the last word on line 302

```
:302    # jump to line 302
$       # jump to the end of the line — cursor is now on the last character
b       # (optional) back to the start of that last word
```

Find the word after the 2nd occurrence of the first word on line 14

```
:14     # jump to line 14
^       # go to the first non-whitespace character (start of first word)
*       # search forward for this word (1st occurrence — which might be itself)
n       # next occurrence (2nd)
w       # move one word forward — now on the word after the 2nd occurrence
```

Commands Introduced in This Chapter

CHAPTER 3 — COMMAND REFERENCE

CHARACTER MOVEMENT

`h / j / k / l`	Left / down / up / right (prefix with count: `5j`)

WORD MOVEMENT

`w`	Forward to start of next word
`W`	Forward to start of next WORD (whitespace-delimited)
`e`	Forward to end of current/next word
`E`	Forward to end of current/next WORD
`b`	Backward to start of current/previous word
`B`	Backward to start of current/previous WORD
`ge`	Backward to end of previous word

LINE MOVEMENT

`0`	Start of line (column 1)
`^`	First non-whitespace character of line
`$`	End of line
`g_`	Last non-whitespace character of line
`f{char}`	Jump forward to next occurrence of char on this line

`F{char}`	Jump backward to previous occurrence of char on this line
`t{char}`	Jump forward to just before char on this line
`T{char}`	Jump backward to just after char on this line
`;`	Repeat last f/F/t/T in the same direction
`,`	Repeat last f/F/t/T in the opposite direction

SENTENCE & PARAGRAPH MOVEMENT

`)`	Forward to start of next sentence
`(`	Backward to start of current/previous sentence
`}`	Forward to next paragraph (blank line)
`{`	Backward to previous paragraph

FILE-LEVEL NAVIGATION

`gg`	Jump to first line of file
`G`	Jump to last line of file
`{n}G`	Jump to line n (e.g. `42G`)
`:{n}`	Jump to line n via command line (e.g. `:302`)
`{n}%`	Jump to n% through the file
`Ctrl+F`	Scroll one full page forward
`Ctrl+B`	Scroll one full page backward
`Ctrl+D`	Scroll half a page down
`Ctrl+U`	Scroll half a page up
`Ctrl+E`	Scroll screen down one line (cursor stays)
`Ctrl+Y`	Scroll screen up one line (cursor stays)
`zz`	Centre current line on screen
`zt`	Scroll current line to top of screen
`zb`	Scroll current line to bottom of screen

SEARCH

`/{pattern}`	Search forward for pattern (Enter to jump)
`?{pattern}`	Search backward for pattern
`n`	Next match (same direction as search)
`N`	Previous match (opposite direction)
`*`	Search forward for exact word under cursor
`#`	Search backward for exact word under cursor

`g*`	Search forward for word under cursor (partial match)
`g#`	Search backward (partial match)
`:noh`	Clear search highlights
`:set hlsearch`	Highlight all search matches
`:set incsearch`	Move cursor as you type the search pattern
`:set ignorecase`	Case-insensitive search
`:set smartcase`	Case-sensitive only when pattern contains uppercase

Learning Vim

Chapter 4 — Changing Text

The Operator + Motion Grammar

In Chapter 3 we built a vocabulary of movement commands. Now we pair those movements with *operators* — commands that act on text — and unlock the composable editing system that makes Vim so powerful.

Almost every editing command in Vim follows one simple grammar:

```
[register] operator [count] motion
```

The register and count are optional. The operator and motion are required.

For example: `d3w` means *delete* (operator) *3 words* (count + motion). The operator and every motion from Chapter 3 can combine freely. Once you have learned the operators in this chapter you effectively have hundreds of editing commands without memorising hundreds of individual keystrokes.

The three core operators are:

Operator	Action	Leaves cursor in
`d`	Delete (cuts to register)	NORMAL
`y`	Yank (copy to register)	NORMAL
`c`	Change (delete + enter Insert mode)	INSERT

Deleting Text

The d operator

The `d` operator deletes the text covered by whatever motion follows it. Critically, deleted text is not just removed — it is placed into a register (Vim's internal clipboard), from where it can be pasted back with `p`. Think of it as "cut" rather than "delete".

```
dw       # delete from cursor to start of next word
de       # delete from cursor to end of current word (leaves trailing space)
db       # delete from cursor back to start of current word
d$       # delete from cursor to end of line
d0       # delete from cursor back to start of line
d^       # delete from cursor back to first non-whitespace char
dG       # delete from current line to end of file
dgg      # delete from current line to start of file
d}       # delete to end of paragraph
d/word   # delete from cursor up to (but not including) the next "word"
3dw      # delete 3 words (count before motion)
d3w      # same — count can go before or after the operator
```

Doubled operators — whole-line shortcuts

Doubling any operator applies it to the entire current line. This is one of the most-used patterns in Vim:

```
dd       # delete the entire current line
5dd      # delete 5 lines starting from the current line
yy       # yank (copy) the entire current line
3yy      # yank 3 lines
cc       # change the entire current line (clears it and enters Insert mode)
```

Shorthand delete commands

A few delete operations are so common they have single-key shortcuts:

Key	Equivalent to	Action
`x`	`dl`	Delete character **under** the cursor
`X`	`dh`	Delete character **before** the cursor

`D`	`d$`	Delete from cursor to **end of line**
`s`	`cl`	Delete character under cursor and enter Insert mode
`S`	`cc`	Delete entire line and enter Insert mode

Before and after — common delete operations

Cursor position shown as the affected region:

Command	Before	After
`dw`	The quick brown fox	The brown fox
`dd`	This entire line is gone	*(line removed)*
`d$`	Hello, World!	Hello,
`x`	Heloo there	Helo there
`xp`	Hello (swap e and l)	Hlelo → *actually:* Hlelo… use `xp` at e → Hello ✓

The xp trick — swapping two characters

A classic Vim idiom for fixing a transposition typo (e.g. "teh" instead of "the") is `xp`: `x` deletes the character under the cursor into the unnamed register, and `p` immediately pastes it after the new cursor position — effectively swapping it with the next character.

```
# Fix "teh" → "the": place cursor on 'e'
x    # deletes 'e', cursor moves to 'h' → "th"
p    # pastes 'e' after 'h' → "the" ✓
```

Undo and redo

Every deletion (and every edit) can be undone. Vim maintains a full undo history for the session:

Key	Action

`u`	Undo the last change
`U`	Undo all changes to the current line (resets the whole line)
`Ctrl+R`	Redo — re-apply an undone change
`{n}u`	Undo the last n changes

> **Vim's undo tree:** Unlike most editors which have a linear undo history, Vim records a full *tree* of changes. If you undo several steps and then make a new edit, the undone branch is not lost — it can be accessed with `:undolist` and navigated with `g-` and `g+`. This is an advanced feature but worth knowing exists.

Yanking and Pasting

The y operator — yank (copy)

"Yank" is Vim's word for copy. The `y` operator works exactly like `d` in terms of motions — the difference is that yanking does not remove the text, it only copies it into a register.

```
yw      # yank from cursor to start of next word
ye      # yank to end of current word
y$      # yank to end of line
y0      # yank from cursor to start of line
yy      # yank entire line (most common)
3yy     # yank 3 lines
yG      # yank from current line to end of file
y}      # yank to end of paragraph
```

The p command — put (paste)

After deleting or yanking, `p` puts the text back. The exact position depends on what was copied:

Key	Behaviour

`p`	Put **after** the cursor. For whole lines, puts below the current line.
`P`	Put **before** the cursor. For whole lines, puts above the current line.
`gp`	Put after the cursor and move the cursor to the end of the pasted text.
`gP`	Put before the cursor and move the cursor to the end of the pasted text.
`{n}p`	Paste n times (e.g. `3p` pastes three copies).

> **Line paste vs character paste:** When you yank or delete an entire line (`yy` , `dd`), Vim marks the register as a *linewise* register. Pasting a linewise register with `p` inserts a new line *below* the current one, regardless of cursor column — very convenient for duplicating or rearranging lines. Character-wise yanks paste inline at the cursor.

Practical workflow — duplicate a line

```
yy    # yank the current line
p     # paste it below — now you have two identical lines

# Or, to duplicate a line and move it somewhere else:
yy    # yank
15j   # move 15 lines down
p     # paste below the new location
```

Visual Mode for Selection-Based Editing

Sometimes the operator + motion model does not map neatly to what you want to select. Visual mode lets you *select text first*, then apply an operator to the selection.

The three visual modes

Key	Mode	Selects
`v`	`VISUAL`	Character by character — any arbitrary range of text

`V`	`V-LINE`	Whole lines — always selects complete lines regardless of column
`Ctrl+V`	`V-BLOCK`	A rectangular block defined by rows and columns

Once in any visual mode, use **any motion command from Chapter 3** to extend the selection. Then press an operator to act on the highlighted region:

```
# Select three lines and delete them:
V         # enter Visual Line mode (current line selected)
2j        # extend selection 2 lines down (3 lines total highlighted)
d         # delete all selected lines

# Select a word and replace it:
viw       # visual select "inner word" (entire word under cursor)
c         # change — deletes selected text and enters Insert mode
# now type the replacement word, then Esc

# Select a paragraph and yank it:
V         # Visual Line mode
}         # extend to end of paragraph
y         # yank the selection
```

Visual Block mode — column editing

Visual Block (`Ctrl+V`) is unique — it selects a rectangular region of the screen, letting you edit multiple lines simultaneously. This is invaluable for tasks like adding a comment character to a block of lines, aligning columns, or deleting a column of whitespace.

```
# Delete the first 4 characters (indentation) from 5 lines:
Ctrl+V    # enter Visual Block
4j        # extend down 4 lines (5 lines total selected)
3l        # extend right to cover 4 columns
d         # delete the selected block

# Add a "#" comment to the start of 4 lines simultaneously:
Ctrl+V    # Visual Block
3j        # select 4 lines
I         # Insert at start of block (capital I)
```

```
#          # type the characters to insert
Esc        # the text is applied to all selected lines
```

Visual Block insert: When you press `I` in Visual Block mode, you only see the text being typed on the first line. After pressing `Esc`, Vim applies the same insertion to every line in the block. This looks like nothing is happening until you press Esc — do not be alarmed.

The Change Operator

The `c` operator is a shortcut for the extremely common pattern of *delete something, then type something new*. It works exactly like `d` but drops you into Insert mode immediately after the deletion, ready to type the replacement.

```
cw       # change word: delete to start of next word, enter Insert mode
ce       # change to end of word (more precise — does not eat trailing space)
c$       # change to end of line (also written C)
c0       # change from cursor to start of line
cc       # change entire line
ciw      # change inner word (deletes entire word, even from middle)
ci"      # change inside quotes: deletes content between " " and enters Insert
ci(      # change inside parentheses: deletes content between ( )
c/text   # change from cursor up to next occurrence of "text"
3cw      # change 3 words
```

Text objects — `iw`, `i"`, `i(` and friends: The `i` in `ciw` stands for *inner* — it selects the word without surrounding whitespace. Its partner `a` (for *around*) includes the surrounding whitespace or delimiters. Text objects work with any operator:

- `diw` — delete inner word
- `ci(` — change inside parentheses
- `yi"` — yank inside double quotes
- `da[` — delete around square brackets (including the brackets)

> Text objects are among the most powerful features in Vim and are worth practising until they feel natural.

Registers — Vim's Multiple Clipboards

Every time you delete or yank text, it goes into a *register*. By default it goes into the *unnamed register* (`"`), which is what `p` pastes from. But Vim has 26 named registers — one for each letter of the alphabet — letting you store multiple pieces of text simultaneously.

Why named registers matter

The problem with relying solely on the unnamed register is that any subsequent delete or yank overwrites it. If you yank some text, then accidentally delete a word before pasting, your yanked text is gone. Named registers solve this:

```
# Copy a function signature to register 'a':
"ayy          # yank current line into register a

# Do other edits (the unnamed register may get overwritten)...
dd            # delete a line (overwrites unnamed register)

# Paste the function signature from register 'a' — still safe:
"ap           # paste from register a
```

Register syntax

```
# General syntax: "x before any operator or paste command
"ayy          # yank line into register a
"bdw          # delete word into register b
"cp           # paste from register c
"Ayy          # APPEND to register a (uppercase = append, lowercase = overwrite)

# View all register contents:
:registers    # (or :reg for short)
```

Special registers

Register	Contents
`"`	The unnamed register — what `p` pastes by default
`0`	The yank register — always holds the last *yanked* (not deleted) text. Useful when a delete has overwritten `"`.
`1` – `9`	Delete history — `1` is the most recent deletion, `2` the one before that, etc.
`+`	The system clipboard — use `"+y` to copy to and `"+p` to paste from the OS clipboard
`*`	The primary selection (Linux/X11 middle-click clipboard)
`_`	The black hole register — delete here and the text is truly gone, not stored anywhere
`/`	The last search pattern
`%`	The current filename

> **The yank register (`0`) is your safety net:** If you yank a line (`yy`), then accidentally delete something before pasting, `"0p` will always paste your last yank — even though the unnamed register was overwritten. Make `"0p` a habit when you have a specific yank you want to preserve.

Search and Replace

Vim's substitute command is one of its most powerful features. It uses the same regular expression engine as the `/` search and can replace text across a range of lines or the entire file.

The substitute command

```
:s/old/new        # replace FIRST occurrence of 'old' on the current line
:s/old/new/g      # replace ALL occurrences on the current line (global flag)
```

```
:s/old/new/gc      # replace all on line, ask for confirmation on each
:s/old/new/gi      # replace all, case-insensitive
```

Specifying a range

```
:%s/old/new/g      # replace all in the ENTIRE FILE (% = all lines)
:5,15s/old/new/g   # replace all between lines 5 and 15
:.,+5s/old/new/g   # replace all from current line (.) to 5 lines below
:'<,'>s/old/new/g  # replace in Visual selection (range is set automatically)
```

Useful substitute patterns

```
# Replace and use the matched text in the replacement (\0 or &):
:%s/error/**&**/g        # wrap every "error" in ** **

# Delete all blank lines:
:%s/^$//g

# Remove trailing whitespace from every line:
:%s/\s\+$//g

# Confirmation flags — responses when using /gc:
#   y = yes (replace this one)
#   n = no (skip this one)
#   a = all (replace all remaining without asking)
#   q = quit (stop substituting)
#   l = last (replace this one, then quit)
```

Spell Checking

Vim has a built-in spell checker that is disabled by default. It is useful for documentation, commit messages, or any prose writing done in Vim.

```
:set spell           # enable spell checking
:set spelllang=en_gb # set language (en_us, en_gb, fr, de, etc.)
```

```
:set nospell        # disable spell checking
```

When spell checking is on, misspelled words are highlighted. Use these keys to navigate and fix them:

Key	Action
`]s`	Jump to the next misspelled word
`[s`	Jump to the previous misspelled word
`z=`	Show a list of suggested corrections for the word under the cursor
`zg`	Mark word as good (add to personal dictionary)
`zw`	Mark word as wrong (add to personal bad-word list)
`1z=`	Accept the first suggestion without showing the list

Commands Introduced in This Chapter

```
CHAPTER 4 — COMMAND REFERENCE

DELETE OPERATOR
d{motion}          Delete text covered by motion (e.g. dw, d$, dG)
dd                 Delete entire current line
{n}dd              Delete n lines
D                  Delete from cursor to end of line (= d$)
x                  Delete character under cursor
X                  Delete character before cursor
xp                 Swap current character with the next (fix transposition typos)

UNDO / REDO
u                  Undo last change
U                  Undo all changes to current line
Ctrl+R             Redo (re-apply an undone change)

YANK (COPY)
```

`y{motion}`	Yank text covered by motion (e.g. `yw`, `y$`, `yG`)
`yy`	Yank entire current line
`{n}yy`	Yank n lines
`Y`	Yank to end of line (some configs) — or `yy` depending on settings
PUT (PASTE)	
`p`	Put (paste) after cursor / below current line
`P`	Put before cursor / above current line
`gp`	Put after cursor, move cursor to end of pasted text
`gP`	Put before cursor, move cursor to end of pasted text
CHANGE OPERATOR	
`c{motion}`	Delete motion + enter Insert mode (e.g. `cw`, `c$`)
`cc`	Change entire current line
`C`	Change from cursor to end of line (= `c$`)
`s`	Substitute character (delete char under cursor + Insert mode)
`S`	Substitute line (delete entire line + Insert mode = `cc`)
`ciw / caw`	Change inner/around word (text object)
`ci" / ca"`	Change inside/around double quotes
`ci( / ca(`	Change inside/around parentheses
VISUAL MODE	
`v`	Enter character-wise Visual mode
`V`	Enter line-wise Visual mode
`Ctrl+V`	Enter block-wise Visual mode (column selection)
REGISTERS	
`"x{op}`	Use named register x for next operation (e.g. `"ayy`, `"ap`)
`"Ax{op}`	Append to register x (uppercase letter)
`"0p`	Paste from yank register (last yank, not overwritten by delete)
`"+y / "+p`	Yank to / paste from the system clipboard
`"_d{motion}`	Delete to black hole register (truly removes, no paste)
`:registers`	Show contents of all registers
SEARCH AND REPLACE	
`:s/old/new`	Replace first occurrence on current line

`:s/old/new/g`	Replace all occurrences on current line
`:s/old/new/gc`	Replace all on current line with confirmation
`:%s/old/new/g`	Replace all occurrences in entire file
`:5,15s/old/new/g`	Replace in line range 5–15
`:'<,'>s/old/new/g`	Replace in Visual selection

SPELL CHECKING

`:set spell`	Enable spell checking
`:set spelllang=en_gb`	Set spell check language
`:set nospell`	Disable spell checking
`]s / [s`	Jump to next / previous misspelled word
`z=`	Show spelling suggestions for word under cursor
`zg / zw`	Add word to personal dictionary / bad-word list

Learning Vim

Chapter 5 — Marks

The Problem Marks Solve

As files grow longer, navigation becomes a two-part problem. The first part — getting to a known location quickly — is solved by search and line-number jumps (Chapter 3). The second part is subtler: *getting back to where you were.*

Imagine you are editing a configuration function on line 340. You need to check how a variable is defined on line 12, then verify a related function on line 580, then return to your original work. Without marks you have to remember three line numbers and type `:340`, `:12`, `:580`, `:340` repeatedly — and if you navigate elsewhere in between, you lose track entirely.

Marks are named bookmarks you can place at any position in a file (or across files). They remember both the line number and the column, and they survive navigation, edits, and even closing and reopening Vim. Combined with the jump list, they form a complete navigation history system.

Setting and Jumping to Marks

Creating a mark

To set a mark, press `m` in Normal mode followed by a single letter — the mark's name:

```
ma    # set mark 'a' at the current cursor position (line + column)
mB    # set mark 'B' (uppercase — works across files)
mz    # set mark 'z'
```

The mark is set silently — no confirmation appears. You can verify it was set with `:marks`.

Jumping to a mark

There are two jump commands, and the difference between them is important:

Command	Jumps to	Use when
`'a` (single quote)	The **beginning of the line** containing mark *a*	You want to work with the whole line
`` `a `` (backtick)	The **exact cursor position** (line *and* column) of mark *a*	You want to return to the precise character

```
# Set a mark, move away, return to it:
ma          # mark current position as 'a'
G           # jump to end of file
'a          # jump back to the LINE containing mark 'a'
`a          # jump back to the EXACT COLUMN of mark 'a'
```

Backtick vs single quote in practice: Most of the time `'a` (single quote) is what you want — it puts you at the start of the relevant line, which is natural for editing. Use `` `a `` (backtick) when column precision matters, such as when returning to a specific character inside a long expression.

Types of Marks

`a – z` · Local marks

Set with `ma` – `mz`. Each mark is local to the buffer (file) it was set in. If you open a different file, `'a` still refers to the position in the original file — it will switch you back to that buffer. You have 26 of these available per file.

`A – Z` · Global marks

Set with `mA` – `mZ`. Global marks work *across files*. Jumping to a global mark with `'A` will open the file containing that mark (if it is not already open) and jump to the position. Ideal for bookmarking key files you switch between frequently.

Automatic marks — set by Vim itself

These are always available without any setup. They are among the most useful marks in daily editing.

Mark	Jump command	What it records
`.`	`'.` or `` `. ``	Position of the **last change** made to the file
`` ` ``	``` `` ```	Position **before the last jump** (a single backtick typed twice)
`'`	`''`	Line of the **last jump** (two single quotes)
`[`	`'[`	Start of the **last changed or yanked text**
`]`	`']`	End of the **last changed or yanked text**
`<`	`'<`	Start of the **last Visual selection**
`>`	`'>`	End of the **last Visual selection**
`^`	`'^`	Position where **Insert mode was last exited**

> **The dot mark `'.` is invaluable.** After making an edit anywhere in the file — even after extensive navigation away — `` `. `` takes you back to the exact character you last changed. No need to remember line numbers. This is one of the first automatic marks to build into muscle memory.

The special ``` `` ``` mark — jump back

The double backtick ``` `` ``` (or double single quote `''` for line-level) is your instant "go back" command. Any time Vim performs a large jump — `G`, `gg`, `/{search}`, `:{line}`, a mark jump — it records your position before the jump. Pressing ``` `` ``` returns you there.

```
G       # jump to end of file
``      # jump back to where you were before pressing G
``      # jump forward again (toggles between the two positions)
```

This toggle behaviour — ``` `` ``` bouncing between two locations — is extremely useful when comparing two distant parts of a file.

Managing Marks

Viewing all marks

```
:marks          # list all marks and their positions
:marks abc      # list only marks a, b, and c
```

The output shows the mark letter, the line number, the column, and a snippet of the text at that position:

```
 mark line  col file/text
  'a    42    8 function processOrder(id) {
  'B   103    0 ~/projects/config.yaml
  '.    28   15 const result = calculateTotal(items)
  ''   100    0 // End of configuration section
```

Deleting marks

```
:delmarks a      # delete mark 'a'
:delmarks a-d    # delete marks a, b, c, and d
:delmarks!       # delete all lowercase marks in the current buffer
```

Using marks with operators

One of the most powerful and underused applications of marks: using them as *motion endpoints* with operators. Any operator (`d` , `y` , `c` , etc.) can take a mark as its motion target:

```
# Yank from current position to mark 'a':
y`a      # yank to exact position of mark a
y'a      # yank to start of line containing mark a (linewise)

# Delete from current line to mark 'b':
d'b

# Classic pattern: mark the start of a block, move to the end, operate:
ma           # mark the start of a function
```

```
/^}          # search for the closing brace on its own line
d'a          # delete everything from mark 'a' to here
```

The Jump List

Marks are for positions you *explicitly* choose to remember. The jump list is different — it is an automatic record of every "large" movement you make, maintained by Vim without any action on your part.

What counts as a jump?

Not every cursor movement is recorded. Vim records a jump when you use:

- File-level navigation: `G`, `gg`, `{n}G`, `:{n}`
- Search: `/pattern`, `?pattern`, `n`, `N`, `*`, `#`
- Mark jumps: `'a`, `` `a ``
- Paragraph/sentence jumps: `{`, `}`, `(`, `)`
- Opening a file: `:e`, `gf`

Character movement (HJKL, `w`, `b`, `e`) does *not* create jump entries.

Navigating the jump list

Command	Action
`Ctrl+O`	Go to the **older** (previous) position in the jump list — jump backward through history
`Ctrl+I`	Go to the **newer** (next) position in the jump list — jump forward through history
`:jumps`	Display the complete jump history list

```
# Example session — the jump list builds up automatically:
:100         # jump to line 100 (recorded)
/function    # search for "function" (recorded)
G            # jump to end of file (recorded)
```

```
Ctrl+O      # back to where /function landed
Ctrl+O      # back to line 100
Ctrl+I      # forward to /function result again
```

> **Ctrl+O / Ctrl+I work across files.** If you jump to a different file with `:e` or `gf` (go to file under cursor), Ctrl+O will take you back to the previous file at the position you left. This makes the jump list a cross-file navigation history, complementing global marks.

Viewing the jump list

```
:jumps
```

```
 jump line  col file/text
   4   340    0 function processOrder(id) {
   3   100    0 // Configuration section
   2    12    4 const BASE_URL = "https://...
   1   580   12 function validateResult(x) {
>  0   340    0 function processOrder(id) {
```

The `>` shows your current position in the list. Numbers on the left show how many `Ctrl+O` presses are needed to reach that entry.

Marks in a Multi-File Workflow

Global marks (A–Z) deserve a dedicated example because they fundamentally change how you work with projects containing multiple files.

Workflow — jumping between project files with global marks

1	`vim app.py`	Open your main application file.
2	`mA`	Set global mark `A` here — "this is my main entry point".
3	`:e config.yaml`	Switch to the config file.
4	`mC`	Set global mark `C` here — "this is the config file".

5	`:e tests/test_app.py`	Switch to the test file.
6	`mT`	Set global mark `T` — "this is the test file".
7	`'A`	Jump instantly back to app.py at the marked position — no `:e` needed.
8	`'C / 'T`	Toggle freely between config and tests. These marks persist across sessions too.

Practical Editing Workflow with Marks

Here is a realistic scenario that ties together marks, the jump list, and the editing commands from earlier chapters.

Editing a Markdown document with a table of contents

Suppose you have a long Markdown file where the table of contents at the top lists section headers, and you want to add content under each header. Marks make this vastly more efficient than scrolling back and forth:

Workflow — table of contents to section content

1	`gg / mT`	Go to the top of the file and set mark `T` on the table of contents — your reference point.
2	`'T + yy`	Return to the TOC and yank a chapter title line you want to work on next.
3	`/^## Chapter 2`	Search for the corresponding section heading in the body of the document.
4	`ms`	Set mark `s` here — "this is the section I'm currently writing".
5	`o + (type content)`	Open a new line below the heading and write the section content.
6	`Esc / 'T`	

		Finished writing? Jump back to the TOC instantly to pick the next section.
7	`` `. (backtick-dot) ``	At any point, jump back to the exact character of the last edit — even if you have navigated far away.

Navigating a large file by line number with marks

```
# Sherlock.txt navigation with marks:
:20       # jump to line 20
ms        # mark it as 's' (start of interesting section)
:100      # jump to line 100
me        # mark it as 'e' (end of section)

# Now operate between the two marks:
's        # jump to mark s
y'e       # yank from here to mark e (linewise — whole lines)
G         # go to end of file
p         # paste the copied section at the end
```

Commands Introduced in This Chapter

CHAPTER 5 — COMMAND REFERENCE

SETTING MARKS

`m{a-z}`	Set a local mark (current buffer only)
`m{A-Z}`	Set a global mark (works across files, persists across sessions)

JUMPING TO MARKS

`'{mark}`	Jump to the **start of the line** containing the mark
`` `{mark} ``	Jump to the **exact position** (line and column) of the mark
`''`	Jump back to line of last jump (toggle between two lines)
`` `` ``	Jump back to exact position before last large jump (toggle)

AUTOMATIC MARKS

`` `. (backtick-dot) ``	Jump to exact position of last change
`'.`	Jump to line of last change

`` `[ and `] ``	Start and end of last changed or yanked text
`` `< and `> ``	Start and end of last Visual selection
`` `^ ``	Position where Insert mode was last exited

MANAGING MARKS

`:marks`	List all current marks
`:marks {letters}`	List specific marks (e.g. `:marks abc`)
`:delmarks {letter}`	Delete a named mark
`:delmarks!`	Delete all lowercase marks in the current buffer

MARKS AS OPERATOR TARGETS

`` y`{mark} ``	Yank from cursor to exact position of mark
`d'{mark}`	Delete from cursor to line of mark (linewise)
`` c`{mark} ``	Change from cursor to exact position of mark

THE JUMP LIST

`Ctrl+O`	Go to older (previous) position in jump list
`Ctrl+I`	Go to newer (next) position in jump list
`:jumps`	Display the full jump history

Learning Vim

Chapter 6 — Buffers, Windows, and Tabs

Three Concepts — One Mental Model

Vim organises its workspace around three distinct but related concepts. Conflating them causes confusion; keeping them clear makes everything else click into place.

`Buffer`

An in-memory copy of a file (or unnamed content). Editing happens here. Nothing is written to disk until you explicitly save.

Think of it as: the document.

`Window`

A viewport into a buffer. You can have many windows open simultaneously, and multiple windows can display the same buffer — changes in one are immediately visible in all.

Think of it as: the pane on screen.

`Tab page`

A collection of windows. Each tab has its own window layout. Tabs are not like browser tabs (one file each) — they are more like desktop workspaces.

Think of it as: a layout preset.

The key insight: **closing a window does not delete a buffer**. If you split a window to view two files and then close one of the windows, both files are still in memory — only the view has been removed. You can return to any buffer in the list at any time.

Buffers

Opening files into buffers

Every file you open in Vim becomes a buffer. Whether you open them from the command line or from within Vim, they all live in the buffer list.

```
# From the shell — open multiple files at once:
vim file1.py file2.py file3.py    # all three become buffers
```

```
# From within Vim — add a new buffer:
:e newfile.txt                    # opens file in current window
:e .                             # open the directory browser
```

Listing buffers

```
:ls        # list all open buffers (also :buffers or :files)
```

The output shows each buffer's number, status flags, name, and current line:

```
  1 %a   "app.py"                line 42
  2  h   "config.yaml"           line 1
  3  h   "tests/test_app.py"     line 18
  4 #a   "utils.py"              line 7
```

Reading the status flags:

Flag	Meaning
`%`	Current buffer (the one displayed in the active window)
`#`	Alternate buffer — the one you were in before switching. `Ctrl+^` toggles between current and alternate.
`a`	Active — loaded and visible in a window
`h`	Hidden — loaded in memory but not currently displayed
`+`	Modified — has unsaved changes
`-`	Not modifiable (read-only)
`=`	Read-only

Switching between buffers

```
:bn            # next buffer in the list (:bnext)
:bp            # previous buffer (:bprevious)
:b 3           # switch to buffer number 3
```

```
:b config       # switch to buffer whose name contains "config" (partial match)
:bf             # first buffer in the list
:bl             # last buffer in the list
Ctrl+^          # toggle between current and alternate (#) buffer — fastest switch
```

Tab completion on buffer names: When typing `:b` you can press `Tab` to cycle through buffer names. Start typing a few characters of the filename first to narrow the list: `:b con<Tab>` might complete to `:b config.yaml`.

Deleting and unloading buffers

```
:bd             # delete (unload) the current buffer
:bd 3           # delete buffer number 3
:bd!            # force-delete even if buffer has unsaved changes
:bw             # wipe buffer — removes it from the list entirely (vs just unloadi
%bd             # delete ALL buffers (useful for a clean slate)
```

`:bd` vs `:bw`: `:bd` (delete) unloads the buffer from memory but keeps it in the buffer list marked as unloaded — you can reload it. `:bw` (wipe) removes it from the list completely, including any marks, the jump list entry, and its history. For most purposes, `:bd` is sufficient.

Reloading a file from disk

```
:e!             # reload current file, discarding all unsaved changes
:e              # reload if file changed on disk (fails if unsaved changes)
```

The directory browser

Opening a directory with `:e .` launches Vim's built-in file browser (**netrw**). You can navigate the filesystem and open files without leaving Vim:

```
:e .            # open current directory in netrw
```

```
:e ~/projects  # open a specific directory
```

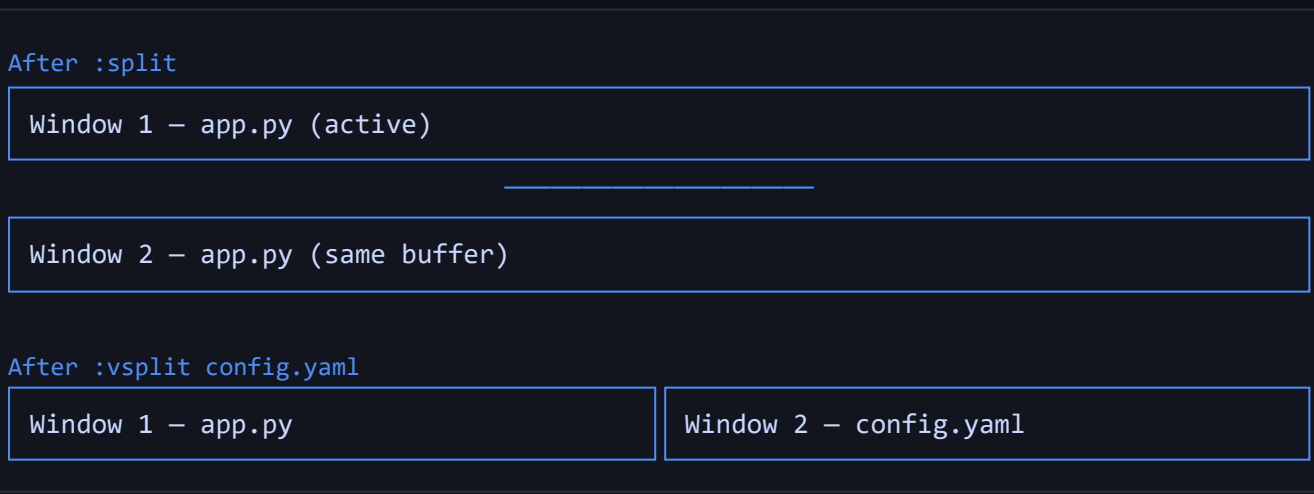
Inside netrw: press `Enter` to open a file or descend into a directory, `-` to go up a level, `D` to delete, `R` to rename, and `%` to create a new file.

Windows — Splitting the Screen

Windows let you view multiple buffers (or multiple positions in the same buffer) simultaneously on screen. This is one of Vim's most productive features for comparing files, referencing documentation while you code, or working on related sections of a large file.

Creating splits

```
:split          # horizontal split — opens another window to the SAME buffer
:split file.py  # horizontal split — opens a DIFFERENT file in the new window
:vsplit         # vertical split (side by side) of same buffer
:vsplit file.py# vertical split opening a different file
Ctrl+W s      # shortcut for :split
Ctrl+W v      # shortcut for :vsplit
Ctrl+W n      # new empty horizontal split
```



Navigating between windows

All window navigation uses the `Ctrl+W` prefix followed by a direction or action key:

Command	Action

`Ctrl+W w`	Cycle to the next window (clockwise)
`Ctrl+W W`	Cycle to the previous window (counter-clockwise)
`Ctrl+W h`	Move to the window to the left
`Ctrl+W j`	Move to the window below
`Ctrl+W k`	Move to the window above
`Ctrl+W l`	Move to the window to the right
`Ctrl+W t`	Move to the top-left window
`Ctrl+W b`	Move to the bottom-right window

> **HJKL for window navigation:** `Ctrl+W h/j/k/l` mirrors the HJKL movement from Chapter 3 — left, down, up, right. Once this mapping clicks it becomes completely natural.

Resizing windows

```
Ctrl+W =      # make all windows equal size
Ctrl+W _      # maximise current window height
Ctrl+W |      # maximise current window width
Ctrl+W +      # increase current window height by 1 line
Ctrl+W -      # decrease current window height by 1 line
Ctrl+W >      # increase current window width by 1 column
Ctrl+W <      # decrease current window width by 1 column
10Ctrl+W +   # increase height by 10 lines (count prefix works)

# Or set an exact size:
:resize 30    # set current window height to 30 lines
:vertical resize 80  # set width to 80 columns
```

Closing and rearranging windows

```
Ctrl+W c      # close current window (buffer remains in list)
Ctrl+W o      # close all OTHER windows — only keep current one (":only")
:q             # quit current window (if only one window, quits Vim)
```

```
:only          # same as Ctrl+W o

# Moving windows:
Ctrl+W r       # rotate windows downward/rightward
Ctrl+W R       # rotate upward/leftward
Ctrl+W x       # exchange current window with the next one
Ctrl+W H       # move current window to far left (makes it a full-height vertical s
Ctrl+W J       # move current window to bottom (full-width horizontal split)
Ctrl+W K       # move current window to top
Ctrl+W L       # move current window to far right
```

Opening a new buffer in a split

```
:sp other.py   # open other.py in a horizontal split
:vsp other.py  # open in a vertical split
:sb 3          # open buffer 3 in a horizontal split
:vert sb 3     # open buffer 3 in a vertical split
```

Tab Pages

Tab pages are a layer above windows. Each tab holds its own set of windows and their layout. They are useful for organising completely separate workspaces — for example, one tab for source files and another for documentation — rather than for switching between individual files (that is what buffers and `Ctrl+^` are for).

```
:tabnew         # open a new empty tab
:tabnew file.py# open file in a new tab
:tabclose       # close the current tab
:tabnext        # go to next tab (also gt in Normal mode)
:tabprevious    # go to previous tab (also gT in Normal mode)
gt              # next tab
gT              # previous tab
2gt             # jump to tab 2
:tabs           # list all tabs and their windows
```

Reading External File Content

The `:r` (read) command inserts the contents of another file directly into the current buffer at the cursor position. This is useful for inserting licence headers, boilerplate code, templates, or any file content you want to pull in without leaving Vim.

```
:r licence.txt  # insert contents of licence.txt BELOW the current line
:0r header.py   # insert at the very top of the file (line 0 = before line 1)
:-1r file.txt   # insert one line ABOVE the cursor
:$r footer.txt  # insert at the very end of the file
:42r snippet.py # insert after line 42 specifically
```

Tab completion works with `:r` just as with `:e` — start typing the path and press `Tab` to complete it.

Reading command output into the buffer

A powerful variant: `:r!` executes a shell command and inserts its output into the buffer:

```
:r! date              # insert today's date at the cursor position
:r! ls *.py           # insert a list of Python files
:r! cat README.md     # insert contents of README (same as :r but via shell)
:r! python3 -c "import sys; print(sys.version)"   # insert Python version string
```

Practical Workflow — Comparing Two Files

A common real-world use case: comparing two versions of a file or checking how a config value is used in two different scripts.

Workflow — side-by-side file comparison

1	`vim app_v1.py`	Open the first file from the shell.
2	`:vsplit app_v2.py`	Open the second file in a vertical split — both are now visible side by side.

3	`Ctrl+W =`	Equalise the window widths so both get the same space.
4	`Ctrl+W h / l`	Move between the two windows using HJKL navigation.
5	`:set scrollbind`	Run in *both* windows (switch with Ctrl+W l, run again). Now scrolling one window scrolls both simultaneously — perfect for comparison.
6	`Ctrl+W o`	When done comparing, close all other windows and return to a single view.

Vim's built-in diff mode: For a proper diff view with changes highlighted, open Vim in diff mode from the shell: `vimdiff file1.py file2.py`. This automatically splits the screen, enables `scrollbind`, and colour-highlights the differences between the two files. Navigate between differences with `]c` (next change) and `[c` (previous change).

Practical Workflow — Adding a Licence Header

Workflow — inserting a licence header with :r

1	`vim script.py`	Open the Python file that needs the header.
2	`:0r licence.txt`	Insert the licence file content at the very top (line 0 = before line 1). Tab complete the filename.
3	`gg`	Jump to the top to verify the header was inserted correctly.
4	`:w`	Save the file.

Commands Introduced in This Chapter

CHAPTER 6 — COMMAND REFERENCE

BUFFER OPERATIONS

`:ls`	List all open buffers (also `:buffers`, `:files`)

`:bn / :bp`	Next / previous buffer
`:bf / :bl`	First / last buffer in the list
`:b {n}`	Switch to buffer number n
`:b {name}`	Switch to buffer matching name (partial, Tab-complete)
`Ctrl+^`	Toggle between current and alternate (#) buffer
`:bd`	Delete (unload) current buffer
`:bd!`	Force-delete buffer with unsaved changes
`:bw`	Wipe buffer (remove from list entirely)
`%bd`	Delete all buffers
`:e!`	Reload current file from disk, discard changes
`:e .`	Open directory browser (netrw)
WINDOW SPLITS	
`:split / :sp`	Horizontal split (same or new buffer)
`:vsplit / :vsp`	Vertical split (side by side)
`Ctrl+W s`	Horizontal split (shortcut)
`Ctrl+W v`	Vertical split (shortcut)
`Ctrl+W n`	New empty horizontal split
`:sb {n}`	Open buffer n in a horizontal split
`:vert sb {n}`	Open buffer n in a vertical split
WINDOW NAVIGATION	
`Ctrl+W w / W`	Cycle to next / previous window
`Ctrl+W h/j/k/l`	Move to window left / down / up / right
`Ctrl+W t / b`	Move to top-left / bottom-right window
WINDOW RESIZING	
`Ctrl+W =`	Equalise all window sizes
`Ctrl+W _ / |`	Maximise current window height / width
`Ctrl+W +/-`	Increase / decrease window height
`Ctrl+W >/<`	Increase / decrease window width
`:resize {n}`	Set window height to n lines
`:vertical resize {n}`	Set window width to n columns
WINDOW CLOSING & REARRANGING	
`Ctrl+W c`	Close current window

`Ctrl+W o / :only`	Close all other windows, keep only current
`Ctrl+W r / R`	Rotate windows down/right or up/left
`Ctrl+W x`	Exchange current window with next
`Ctrl+W H/J/K/L`	Move window to far left/bottom/top/right

TAB PAGES

`:tabnew / :tabnew {file}`	Open a new tab (empty or with file)
`:tabclose`	Close current tab
`gt / gT`	Go to next / previous tab
`{n}gt`	Go to tab number n
`:tabs`	List all tabs and their windows

READING FILES

`:r {file}`	Insert file contents below current line
`:0r {file}`	Insert at very top of file
`:-1r {file}`	Insert one line above cursor
`:{n}r {file}`	Insert after line n
`:r! {cmd}`	Insert output of shell command into buffer

DIFF MODE

`vimdiff f1 f2`	Open two files in diff mode (from shell)
`:set scrollbind`	Link scrolling of multiple windows together
`]c / [c`	Jump to next / previous change in diff mode

Learning Vim

Chapter 7 — Configuration

Why Configuration Matters

Out of the box, Vim is a capable but deliberately spartan editor. Its default settings prioritise compatibility with ancient systems over convenience for modern workflows. The good news: Vim's configuration system is among the most powerful of any editor — you can remap any key, define your own commands, add abbreviations, and change dozens of behaviours, all from a single plain-text file.

This chapter builds a foundational `.vimrc` — the configuration file Vim reads on startup — that turns the defaults into something genuinely comfortable for daily use.

The .vimrc File

Location

System	Location
macOS / Linux	`~/.vimrc` (a hidden file in your home directory)
Windows	`C:\Users\yourlogin\.vimrc` or `C:\Users\yourlogin\_vimrc`
Vim 8+ (any)	`~/.vim/vimrc` (no dot, inside the `.vim` directory) — preferred modern location

If the file does not exist, create it. Vim reads it automatically every time it starts.

Syntax basics

The `.vimrc` is a plain text file containing Vim commands, one per line — exactly as you would type them at the `:` prompt, but without the leading colon. Comments begin with `"` (a double-

quote character).

```
" This is a comment — anything after " on a line is ignored
set number " same as :set number at the command prompt
syntax on " same as :syntax on
```

Reloading your config without restarting

After editing `.vimrc` you can apply changes immediately without closing Vim:

```
:source ~/.vimrc    # reload the vimrc file
:so %               # if you are currently editing vimrc, source the current file
```

> **The portability trade-off:** A heavily customised Vim feels fast and natural on your own machine, but when you SSH into a server or pair program on someone else's machine, you are working with vanilla Vim again. The advice to keep your config lean is not about limiting power — it is about ensuring you stay comfortable in stock Vim. Aim for a config that enhances rather than replaces Vim's native behaviour.

Essential Settings

These are the settings most Vim users consider non-negotiable. Add them to your `.vimrc` and they will apply every time Vim opens.

```
" ── Sensible defaults ──────────────────────────────
set nocompatible " disable Vi compatibility mode — unlock all Vim features
filetype on " detect file type for syntax and indentation rules
filetype plugin on " load filetype-specific plugins
filetype indent on " load filetype-specific indentation rules

" ── Display ────────────────────────────────────────
set number " show line numbers (abbrev: se nu)
set relativenumber " show relative line numbers alongside absolute
set cursorline " highlight the line the cursor is on
```

```
set wrap " wrap long lines (default); set nowrap to disable
set showmatch " briefly highlight matching bracket when cursor is on one
set showcmd " show partial command in status bar as you type
set ruler " show line/column in status bar
syntax on " enable syntax highlighting

" ── Search ──────────────────────────────────────────
set ignorecase " case-insensitive search (abbrev: ic)
set smartcase " override ignorecase when pattern has uppercase
set hlsearch " highlight all search matches
set incsearch " show matches as you type the pattern

" ── Indentation ─────────────────────────────────────
set tabstop=4 " a Tab character is displayed as 4 spaces wide
set shiftwidth=4 " number of spaces used for indentation (>> and <<)
set expandtab " insert spaces instead of tab characters
set autoindent " copy indent from previous line when starting a new one
set smartindent " smart auto-indentation for code blocks

" ── Clipboard ───────────────────────────────────────
set clipboard=unnamedplus " use system clipboard for all yank/delete/paste

" ── Behaviour ───────────────────────────────────────
set hidden " allow switching buffers without saving first
set confirm " ask instead of silently failing on :q with unsaved changes
set history=1000 " remember 1000 commands in history (default is 20)
set undolevels=1000 " allow 1000 levels of undo
set backspace=indent,eol,start " make Backspace work as expected in Insert mode
set wildmenu " enhanced Tab completion menu for commands
set wildmode=list:longest " complete to longest common match, then list options
```

> `set nocompatible` — **always put this first.** Without it, Vim runs in a compatibility mode designed to behave like the original `vi`, which disables many features introduced in Vim. Place it as the very first line of your `.vimrc`.

Checking and toggling settings

```
:set                  # show all non-default settings currently active
:set all              # show every setting with its current value
:set number?          # query the current value of a specific setting
:set nonu             # disable a boolean setting by prepending "no"
:set nu!              # toggle a boolean setting with !
:help option-summary  # browse the full list of all settings in Vim's built-in hel
```

Colour Schemes and Syntax Highlighting

```
:syntax on            # enable syntax colouring (add to .vimrc without the colon)
:syntax off           # disable syntax colouring
:colorscheme desert   # switch to the 'desert' colour scheme
:colorscheme morning  # light theme
:colorscheme delek    # another built-in option
:colorscheme <Tab>    # Tab through all installed colour schemes
```

To list all available schemes: `:colorscheme <Space><Tab>` — this opens the wildmenu showing all options.

To make your chosen scheme permanent, add this to `.vimrc`:

```
colorscheme desert
```

> **Installing additional colour schemes:** Place `.vim` colour files in `~/.vim/colors/`. Hundreds of schemes are available online. Popular choices include *gruvbox*, *molokai*, *solarized*, and *nord*. With a plugin manager (see the tip at the end of this chapter), installing them is a one-liner.

Key Mappings

Key mappings let you reassign any key (or key sequence) to any action. This is one of the most powerful customisation tools in Vim — the right mappings can eliminate dozens of keystrokes

from your daily workflow.

Mapping commands

Command	Mode	Use
`nnoremap`	Normal	Map a key in Normal mode (recommended)
`inoremap`	Insert	Map a key in Insert mode
`vnoremap`	Visual	Map a key in Visual mode
`noremap`	Normal + Visual + Operator	Map in multiple modes at once
`nmap` / `map`	Normal	Map without the "no-remap" safety — **avoid**

> **Always use `noremap` variants, never plain `map`.** The difference is that `noremap` (no-remap) prevents recursive mappings — if you map key A to key B, and later map key B to something else, `noremap` still runs the original action of B. Plain `map` would chain them, causing hard-to-debug loops. Use `nnoremap` for Normal mode, `inoremap` for Insert mode.

Mapping syntax

```
" Syntax: noremap {lhs} {rhs}
" lhs = key to press, rhs = what it does

" Special key notation:
"   <Space>  = spacebar
"   <CR>     = Enter (Carriage Return)
"   <Esc>    = Escape
"   <Tab>    = Tab
"   <C-f>    = Ctrl+f
"   <leader> = your leader key (see below)
```

Practical mappings to add to .vimrc

```
" ── Key mappings ──────────────────────────────────────

" Space scrolls forward a page (like PgDn)
nnoremap <Space> <C-f>

" Tab jumps to the next sentence in Normal mode
nnoremap <Tab> )

" jj in Insert mode acts as Escape — keeps fingers on home row
inoremap jj <Esc>

" Clear search highlights with Enter in Normal mode
nnoremap <CR> :nohlsearch<CR>

" Save with Ctrl+s in Normal and Insert mode
nnoremap <C-s> :w<CR>
inoremap <C-s> <Esc>:w<CR>a

" Move between windows with Ctrl+direction (no need for Ctrl+W prefix)
nnoremap <C-h> <C-w>h
nnoremap <C-j> <C-w>j
nnoremap <C-k> <C-w>k
nnoremap <C-l> <C-w>l

" Navigate buffers quickly
nnoremap <F2> :bprevious<CR>
nnoremap <F3> :bnext<CR>
```

The leader key

The *leader key* is a namespace prefix for your own custom mappings. By convention it is set to `\` (backslash) by default, but most users change it to `,` (comma) or `<Space>` for easier reach. It prevents your mappings from clashing with Vim's built-in commands.

```
" Set leader key to comma
let mapleader = ","
```

```
" Now define leader mappings — press , then the letter
nnoremap <leader>w :w<CR> " ,w = save file
nnoremap <leader>q :q<CR> " ,q = quit
nnoremap <leader>e :e ~/.vimrc<CR> " ,e = open vimrc for editing
nnoremap <leader>s :source ~/.vimrc<CR> " ,s = reload vimrc
```

Abbreviations

Abbreviations (`:ab` or `:iabbrev`) expand short trigger strings into longer text when you type them in Insert mode followed by a space, Enter, or punctuation. They are useful for fixing common typos, expanding jargon, and inserting frequently used phrases.

```
" In the command line (interactive, not in .vimrc):
:ab _me philip.osztromok@email.com   # type _me + space → expands
:ab recieve receive                  # auto-correct typo
:ab _ys Youngster                     # shorthand for a word
```

```
" ── Abbreviations in .vimrc ──────────────────────────

" Typo corrections
iabbrev recieve receive
iabbrev teh the
iabbrev dont don't
iabbrev occured occurred
iabbrev seperate separate

" Quick expansions — underscore prefix avoids accidental triggers
iabbrev _em yourname@example.com
iabbrev _sig Kind regards,<CR>Your Name
iabbrev _dt <C-r>=strftime('%Y-%m-%d')<CR> " inserts today's date
```

Preventing expansion: If you genuinely need to type an abbreviation trigger without expanding it, press `Ctrl+V` immediately before the space or Enter that would trigger expansion. This inserts a literal space without triggering the abbreviation.

> **Use the underscore prefix convention:** Prefixing your personal abbreviations with `_` (e.g. `_me`, `_sig`) means they can never conflict with ordinary words — no English word starts with an underscore. This is a widely followed convention among Vim users.

Custom Commands

The `:command` statement (abbreviated `:com`) lets you define your own Command-mode commands. This is perfect for wrapping external tools you run frequently.

```
" Syntax: :command! CommandName action
" ! overwrites if it already exists
" Command names MUST start with an uppercase letter

:command! PY  !python3 %     # run current Python file (:PY to execute)
:command! WC  !wc %          # word count of current file
:command! LS  !ls -la        # list directory contents
```

The `%` in these commands is a special placeholder that expands to the current filename. So `:PY` runs `python3 app.py` (or whatever the current file is named).

```
" ── Custom commands ──────────────────────────
command! PY !python3 %
command! WC !wc -w %
command! TODAY r!date +\%Y-\%m-\%d " insert today's date below cursor
command! SH !bash % " run current file as a shell script
```

Running external commands interactively

Without defining a custom command, you can run any shell command with `:!`:

```
:!ls              # list directory contents
:!python3 %       # run current file (% = current filename)
:!git status      # check git status without leaving Vim
:!git diff %      # diff current file against last commit
:!chmod +x %      # make current file executable
```

After the command finishes, press `Enter` to return to Vim. The output is displayed in a full-screen shell overlay.

A Complete Starter .vimrc

Here is a well-commented, complete starter configuration you can use as a base and adapt to your own preferences. Every line is explained.

```
" ═══════════════════════════════════════════════════════
" .vimrc — starter configuration
" ═══════════════════════════════════════════════════════

" Must be first — disable Vi compatibility, enable all Vim features
set nocompatible

" ── File type detection ──────────────────────────────
filetype plugin indent on

" ── Appearance ───────────────────────────────────────
syntax on
colorscheme desert " change to your preferred scheme
set number " absolute line numbers
set relativenumber " relative numbers alongside absolute
set cursorline " highlight current line
set showcmd " show partial commands in status bar
set ruler " show line/col in status bar
set showmatch " highlight matching brackets

" ── Search ───────────────────────────────────────────
set ignorecase " case-insensitive search
set smartcase " case-sensitive when pattern has uppercase
set hlsearch " highlight all matches
set incsearch " incremental search as you type

" ── Indentation ──────────────────────────────────────
set tabstop=4 " tab width
set shiftwidth=4 " indent width
set expandtab " spaces instead of tabs
```

```
set autoindent " copy indent from previous line

" ── Behaviour ──────────────────────────────────────
set hidden " allow unsaved buffers to be hidden
set backspace=indent,eol,start" sensible backspace in Insert mode
set history=1000 " command history size
set undolevels=1000 " generous undo history
set wildmenu " enhanced command completion
set wildmode=list:longest " completion behaviour
set clipboard=unnamedplus " use system clipboard

" ── Leader key ─────────────────────────────────────
let mapleader = ","

" ── Key mappings ───────────────────────────────────
nnoremap <Space> <C-f> " Space = page forward
nnoremap <CR> :nohlsearch<CR> " Enter clears search highlights
inoremap jj <Esc> " jj exits Insert mode
nnoremap <C-h> <C-w>h " Ctrl+direction to move between windows
nnoremap <C-j> <C-w>j
nnoremap <C-k> <C-w>k
nnoremap <C-l> <C-w>l
nnoremap <leader>e :e ~/.vimrc<CR> " ,e = edit vimrc
nnoremap <leader>s :source ~/.vimrc<CR> " ,s = reload vimrc

" ── Abbreviations ──────────────────────────────────
iabbrev recieve receive
iabbrev teh the
iabbrev dont don't

" ── Custom commands ────────────────────────────────
command! PY !python3 %
command! WC !wc -w %
```

The Help System

Vim has a comprehensive built-in help system that is worth using. Every setting, command, and mapping is documented.

```
:help                  # open the help index
:help option-summary   # list of all settings with brief descriptions
:help :map             # help for the map command
:help set              # help for the set command
:help normal           # help on Normal mode commands
:help vimrc            # help on the vimrc file itself
:help insert.txt       # full Insert mode documentation
```

Inside the help window, use `Ctrl+]` to follow a hyperlink (shown in `|pipes|`) and `Ctrl+O` to go back. Close the help window with `:q`.

> **Plugin managers:** Once you are comfortable with the basics, a plugin manager dramatically simplifies adding third-party tools. The most popular options are **vim-plug** (minimal, fast), **Vundle** (classic), and **lazy.nvim** (for Neovim). A single line in your `.vimrc` can install colour schemes, file trees, Git integration, and language servers — without cluttering your config.

Commands Introduced in This Chapter

CHAPTER 7 — COMMAND REFERENCE

VIMRC

`:source ~/.vimrc`	Reload the vimrc without restarting Vim
`:so %`	Source the current file (when editing vimrc)

SETTINGS

`:set`	Show all non-default settings currently active
`:set all`	Show every setting with its current value
`:set {option}?`	Query the current value of an option
`:set no{option}`	Disable a boolean option
`:set {option}!`	Toggle a boolean option on/off
`set nocompatible`	Disable Vi compatibility (always first in vimrc)
`set number / nonu`	Show / hide line numbers
`set relativenumber`	Relative line numbers
`set cursorline`	Highlight the current line

`set showmatch`	Highlight matching brackets
`set showcmd`	Show partial commands in status bar
`set hidden`	Allow switching buffers without saving first
`set wildmenu`	Enhanced Tab-completion menu
`set tabstop=4`	Tab display width
`set shiftwidth=4`	Indent width for >> and <<
`set expandtab`	Insert spaces instead of tab characters
`set clipboard=unnamedplus`	Use system clipboard by default
`set history=1000`	Increase command history size
`set undolevels=1000`	Increase undo history depth
`:help option-summary`	Browse all available settings in built-in help

COLOUR SCHEMES & SYNTAX

`:syntax on / off`	Enable / disable syntax highlighting
`:colorscheme {name}`	Switch to a colour scheme
`:colorscheme <Tab>`	Browse all available colour schemes

KEY MAPPINGS (VIMRC)

`nnoremap {key} {action}`	Map a key in Normal mode (safe, non-recursive)
`inoremap {key} {action}`	Map a key in Insert mode
`vnoremap {key} {action}`	Map a key in Visual mode
`let mapleader = ","`	Set the leader key to comma

ABBREVIATIONS

`:ab {trigger} {expansion}`	Define an abbreviation (interactive)
`iabbrev {trigger} {expansion}`	Define an abbreviation in vimrc
`Ctrl+V (in Insert)`	Prevent the next character from triggering an abbreviation

CUSTOM COMMANDS & SHELL

`:command! {Name} {action}`	Define a custom Command-mode command
`:! {shell-command}`	Run a shell command from within Vim
`% (in :! commands)`	Expands to the current filename

HELP

`:help`	Open the help index
`:help {topic}`	Open help for a specific topic or setting

`Ctrl+] (in help)`	Follow a help hyperlink

Learning Vim

Chapter 8 — Day to Day Vim

Introduction

Vim is not just a text editor — it is a power tool for working with files, diffs, archives, and external programs, all without leaving the keyboard. This chapter covers the features you will reach for every day once you know they exist: startup options that automate editing tasks, a proper understanding of diff mode, editing files inside archives, jumping to files by name, and piping buffer content through external programs.

Individually, none of these is complex. Together they turn Vim into a surprisingly versatile part of your shell workflow.

Command-Line Startup Options

When you open Vim from the command line you can pass instructions that run automatically the moment the file loads. This is Vim's scripting entry point — a way to drive the editor without interactive input.

The + prefix: jump to a line or pattern

Any argument beginning with `+` is a command Vim executes after opening the file.

```
# Open a file at a specific line number
vim hello.go +8            # cursor lands on line 8
vim notes.txt +42          # jump straight to line 42
vim report.md +$           # open at last line of file

# Search on open — jump to the first match
vim sherlock.txt +/crumb   # positions cursor at first "crumb"
vim app.log +/ERROR        # jump to first ERROR in a log file
vim config.yaml +/host    # find the "host" key immediately
```

Why this matters for compiler errors: Most build systems print errors as `filename:line: message`. You can open the offending file directly at the error line from the shell: `vim src/main.c +47` — no scrolling required.

Multiple + commands: automated editing

You can chain several `+` commands. They execute in order, left to right, before you see the file. This makes Vim usable as a one-liner scriptable editor in shell pipelines.

```
# Delete lines 1–2 (header rows) then save and quit
vim +"1,2d" +wq conway.txt

# Delete line 4 and save
vim +4d +wq time.txt

# Replace all occurrences in a file without opening it interactively
vim +"%s/localhost/prod.server.com/g" +wq config.txt

# Append a line to the end of a file
vim +"normal G" +"put ='# end of file'" +wq notes.md
```

When to use this vs. `sed`: For simple one-off substitutions `sed -i` is often shorter. But for multi-step operations (delete a range, reformat, save), or when you want to preview the result first, Vim's `+` chain gives you the full power of the editor's command language in a single shell command.

Other useful startup flags

Flag	Effect
`vim +{n} file`	Open at line n
`vim +/{pattern} file`	Open and jump to first match of pattern
`vim -R file`	Read-only mode — prevents accidental edits
`vim -d file1 file2`	Diff mode — compare two files side by side

`vim -u NONE file`	Skip vimrc — useful for diagnosing config issues
`vim -n file`	No swap file — faster for read-only browsing
`vim -c "{cmd}" file`	Equivalent to `+"{cmd}"` — run command after open

Diff Mode

Diff mode is one of Vim's most powerful features and frequently overlooked. It opens two (or more) files in synchronised split windows, highlights every difference, and lets you move changes between files with single-key commands.

Opening diff mode

```
# From the command line — the most common way
vim -d road1.txt road2.txt        # horizontal split by default
vimdiff road1.txt road2.txt        # alias — identical to vim -d
vimdiff v1.py v2.py v3.py          # three-way diff is possible

# From inside Vim, with a file already open
:diffsplit road2.txt               # open road2 below in diff mode
:vert diffsplit road2.txt          # open road2 to the right (vertical)
```

Reading the diff display

```
VIMDIFF ROAD1.TXT ROAD2.TXT
road1.txt                 | road2.txt
                          |
 Two roads diverged       | Two roads diverged
- in a yellow wood,       |+ in a yellow wood
                          | and looked down one
 And sorry I could not    | And sorry I could not
```

Highlighted lines are differences. Fold-collapsed sections are identical text. The vertical bar separates the two windows.

Navigating between differences

```
]c          # jump to next difference (change)
[c          # jump to previous difference
Ctrl+W w    # move to the other diff window
Ctrl+W h/l # move left/right between windows explicitly
```

Applying changes between files

This is the heart of diff mode. You can pull a change from the other file (*obtain*) or push your version to the other file (*put*) without copying and pasting.

```
do          # diff Obtain — pull change from the OTHER file into THIS one
dp          # diff Put    — push change from THIS file into the OTHER one
```

```
HOW DO / DP WORK
You have two windows open side by side. Your cursor is on a highlighted difference.

do → copies the OTHER window's version of this diff block into YOUR window
dp → copies YOUR window's version of this diff block into the OTHER window

Think: "do = take" and "dp = give".
```

Updating the diff after manual edits

```
:diffupdate     # recalculate the diff (if the highlighting looks stale)
:set diffopt+=iwhite   # ignore whitespace-only differences
```

Making vertical diff the default

By default `vimdiff` opens a horizontal split. Most people find a vertical (side-by-side) layout easier to read. Add this to your `.vimrc`:

```
" In .vimrc — always use vertical (side-by-side) diff layout
```

```
set diffopt+=vertical
```

> **Three-way merges:** `vimdiff` supports up to eight files simultaneously. Git's `mergetool` integration uses three-way diff (local / base / remote) when you run `git mergetool` with Vim configured as the merge tool. This is a genuinely useful alternative to GUI merge tools once you are comfortable with `do` and `dp`.

Editing Files Inside Archives

Vim can open and edit files inside ZIP and TAR archives directly — without you needing to extract them first. This works through Vim's *netrw* and archive plugins, which are bundled with a standard Vim installation.

Opening a ZIP file

```
vim weather.zip        # shows a directory listing of the archive's contents
```

```
VIM WEATHER.ZIP — WHAT YOU SEE
" zip.vim version v28
" Browsing zipfile /home/user/weather.zip
" Select a file with cursor and press ENTER

weather/
weather/forecast.txt
weather/historical.csv
weather/README.md
```

- Navigate the listing with `j` / `k`
- Press `Enter` to open a file for editing
- Edit it as normal
- Save with `:w` — changes are written back into the ZIP
- Press `-` or `:bprevious` to return to the archive listing

Platform requirements: On Linux and macOS, the `zip` and `unzip` utilities must be available in your PATH. They almost always are on macOS and most Linux distributions. On Windows, ensure that a ZIP utility is installed and accessible from the command line (Git Bash or WSL both provide this automatically).

TAR archives

The same workflow applies to `.tar`, `.tar.gz`, and `.tar.bz2` files. Vim detects the archive type automatically from the file extension and delegates to the appropriate utility.

```
vim project.tar.gz     # shows contents of the tar archive
vim backup.tar.bz2     # works the same way
```

When is this useful? Typical use cases: quickly checking what is inside a downloaded archive, editing a config file buried in a deployment ZIP, or patching a single file in a release tarball without unpacking the entire thing. It is a surprisingly handy trick that most editors cannot match.

File Navigation: gf and gF

The `gf` command is deceptively simple and enormously useful: it opens whatever filename or path is currently under the cursor. If you are reading code that imports another module, reviewing a config that references a path, or looking at a compiler error, `gf` jumps straight there.

gf — go to file

```
# With cursor on the filename in any of these contexts:

# Source code imports
require('./utils/helper')    # cursor on helper → gf opens the file
import "config/settings.go"  # gf opens settings.go
include "auth.h"             # gf opens auth.h
```

```
# Log and error output
main.py:47: NameError: name 'foo' not defined
# cursor on main.py → gf opens main.py
```

```
gf        # open file under cursor (replaces current window)
Ctrl+W f # open file under cursor in a NEW horizontal split
Ctrl+W gf# open file under cursor in a NEW TAB
```

> **Return from gf:** After jumping with `gf`, press `Ctrl+O` to return to your previous position (it is a jump, so it lands in the jump list).

gF — go to file at line

`gF` extends `gf` — it also reads a line number from the text under the cursor. This is the format that compilers, linters, and most error-reporting tools produce.

```
# Compiler / linter error format:  filename:line:column: message
src/parser.c:83:12: error: expected ';' before '}'
# gF on src/parser.c:83:12 → opens parser.c AND moves cursor to line 83

# Also works with just filename:line
app.py:47
# gF jumps to app.py line 47

# Or with :line syntax written inline
config/database.yml:2
# gF opens database.yml at line 2
```

Helping Vim find files: path setting

By default Vim looks for files relative to the current directory. If your imports use bare names without paths (common in C and Python projects), add the project source directories to Vim's `path` so `gf` can find them:

```
" In .vimrc or set interactively:
:set path+=src/**        # search src/ and all subdirectories
```

```
:set path+=include/**    # also search include/
```

URL fetching with gf

If `wget` or `curl` is installed, Vim can use `gf` to fetch and display a URL directly in the buffer. With your cursor on a URL like `https://example.com/data.json`, pressing `gf` will download and display the content. This requires the `netrw` plugin to be active (it is by default).

```
# Cursor on any URL in the buffer:
https://raw.githubusercontent.com/user/repo/main/README.md
# gf → downloads and displays the file content in Vim
```

External Command Integration

Vim's relationship with the shell goes beyond just running commands and viewing output. You can insert shell output directly into a buffer, and — crucially — you can pipe your entire buffer (or a range of lines) through an external program and replace it with the result. This makes Vim a composable part of your command-line toolkit.

Running shell commands from Vim

```
:!date           # display the current date/time in a shell overlay
:!ls -la         # list directory contents
:!git status     # check git status without leaving Vim
:!python3 %      # run the current file (% = current filename)
:!make           # build the project
```

The output appears in a full-screen shell overlay. Press `Enter` to return to your buffer. Nothing is inserted into the file.

Reading command output into the buffer

The `:r !` form (*read from command*) inserts the stdout of a shell command below the cursor. This is for when you want the output to become part of the file.

```
:r !ls                        # insert file listing below cursor
:r !date                      # insert current timestamp
:r !ping -c 3 www.vim.org     # insert ping results
:r !cat /etc/hosts            # insert hosts file content
:r !curl -s https://api.example.com/data  # fetch and insert API response

# Specify where to insert:
:0r !date                     # insert at top of file (before line 1)
:$r !date                     # insert at end of file
```

Filtering buffer content through external programs

This is the most powerful form of external integration. The `:%!` command sends your entire buffer as *stdin* to an external program, then **replaces** the buffer with the program's stdout. You are using the external tool as a transformer — a formatter, a converter, a filter.

```
# Format / pretty-print
:%!jq .           # format JSON with proper indentation (requires jq)
:%!python3 -m json.tool  # format JSON with Python (no extra install)
:%!tidy -xml -i  # format XML/HTML with tidy
:%!xmllint --format -  # format XML with xmllint

# Sort and deduplicate
:%!sort           # sort all lines alphabetically
:%!sort -u        # sort and remove duplicate lines
:%!uniq           # remove consecutive duplicate lines

# Binary / hex editing
:%!xxd            # convert buffer to hex dump view
:%!xxd -r         # convert hex dump back to binary (after editing)

# Text processing
:%!tr 'a-z' 'A-Z'  # convert all text to uppercase
:%!wc -l          # replace buffer with just the line count
```

Filtering a range, not the whole file

You can filter just a selection of lines. This is useful for sorting a block, formatting a specific JSON object, or processing only part of a file.

```
# Range-based filtering:
:5,15!sort      # sort only lines 5 through 15
:'<,'>!sort    # sort the current Visual selection
:'<,'>!jq .   # format the selected JSON fragment
```

In Visual mode, after selecting lines with `V`, type `!` and Vim will automatically fill in `:'<,'>!` — just complete the command.

> `:%!` **is destructive — back up first.** If the external command fails or produces unexpected output, the buffer is replaced with the error message or empty output. Fortunately, this is a single undoable change — press `u` immediately to restore the original content. Always check your command works on a test input before running it on a file you care about.

The xxd hex editing workflow

The combination of `:%!xxd` and `:%!xxd -r` gives you a genuine hex editor inside Vim. This is how binary files (executables, images, compiled bytecode) can be edited:

Binary editing with xxd

1	`vim binaryfile.bin`	Open the binary file
2	`:%!xxd`	Convert to hex dump — you now see hex values and ASCII on the right
3	Edit the hex values directly (only edit the hex column, not the ASCII column)	
4	`:%!xxd -r`	Convert the hex dump back to binary
5	`:w`	Save — the file is written as binary

Putting It All Together

Practical workflow: fix a compiler error

From error message to fix without leaving the terminal

1	`make 2>&1 | tee errors.txt`	Build and save compiler output to a file
2	`vim errors.txt`	Open the error log in Vim
3	Move cursor to the filename:line in an error line	
4	`gF`	Jump directly to the file at the exact error line
5	Fix the error	
6	`:!make`	Rebuild without leaving Vim
7	`Ctrl+O`	Return to errors.txt to check the next error

Practical workflow: review and merge a diff

Compare two versions and merge selectively

1	`vimdiff old.conf new.conf`	Open both files in diff mode side by side
2	`]c`	Jump to the first difference
3	Review the change — decide which version to keep	
4	`do`	Obtain — accept the other file's version here
5	`dp`	Put — push your version to the other file instead
6	`]c`	Move to the next difference and repeat
7	`:wqa`	Save all and quit when done

Commands Introduced in This Chapter

`CHAPTER 8 — COMMAND REFERENCE`

STARTUP FLAGS	
`vim +{n} file`	Open file at line n
`vim +/{pattern} file`	Open file and jump to first match of pattern
`vim +"{cmd}" file`	Run a command after opening the file
`vim -d file1 file2`	Open two files in diff mode
`vim -R file`	Open file in read-only mode
`vim -u NONE file`	Open without loading vimrc
`vimdiff file1 file2`	Alias for vim -d
DIFF MODE	
`:diffsplit {file}`	Open file in diff mode (horizontal split)
`:vert diffsplit {file}`	Open file in diff mode (vertical split)
`]c`	Jump to next difference
`[c`	Jump to previous difference
`do`	Diff Obtain — pull change from the other file into this one
`dp`	Diff Put — push change from this file into the other one
`:diffupdate`	Recalculate and refresh the diff highlights
`set diffopt+=vertical`	Always use vertical (side-by-side) diff layout (vimrc)
`set diffopt+=iwhite`	Ignore whitespace differences in diff
ARCHIVE EDITING	
`vim {file}.zip`	Browse and edit files inside a ZIP archive
`vim {file}.tar.gz`	Browse and edit files inside a TAR archive
FILE NAVIGATION	
`gf`	Open file whose name is under the cursor
`gF`	Open file under cursor at the line number also under cursor
`Ctrl+W f`	Open file under cursor in a new horizontal split
`Ctrl+W gf`	Open file under cursor in a new tab
`:set path+=src/**`	Add directories to the search path for gf
EXTERNAL COMMANDS	
`:! {cmd}`	Run a shell command; show output in overlay
`:r ! {cmd}`	Insert shell command output below cursor
`:0r ! {cmd}`	Insert shell command output at top of file

`:%! {cmd}`	Filter entire buffer through external command (replaces content)
`:{range}! {cmd}`	Filter a line range through external command
`'<,'>! {cmd}`	Filter Visual selection through external command
`:%!jq .`	Format JSON with jq
`:%!sort`	Sort all lines alphabetically
`:%!xxd`	Convert buffer to hex dump
`:%!xxd -r`	Convert hex dump back to binary

Learning Vim

Appendix A — Going Further

What You Have Covered

This course has taken you from opening a file for the very first time all the way through marks, registers, diff mode, configuration, and shell integration. That is a solid foundation — you now have the mental model that most long-term Vim users rely on every day.

`Chapter 1` Introduction — modal editing, installation, first steps

`Chapter 2` Basics — the five modes, file operations, quit variants

`Chapter 3` Moving Around — HJKL, word and line motions, search

`Chapter 4` Changing Text — operators, yank/put, registers, substitution

`Chapter 5` Marks — local and global marks, the jump list

`Chapter 6` Buffers, Windows and Tabs — splits, vimdiff, reading files

`Chapter 7` Configuration — .vimrc, mappings, abbreviations, custom commands

`Chapter 8` Day to Day Vim — startup flags, diff mode, archive editing, shell filters

> **The most important thing to do next:** keep using it. The hardest part of Vim is not learning the commands — it is building the muscle memory. Commit to using Vim for real work for two weeks without switching back to a comfort-zone editor, and most of it will become automatic.

Built-in Resources You Already Have

:help — the manual that ships with Vim

Vim's built-in help is one of the best pieces of documentation packaged with any open-source tool. It is comprehensive, searchable, and always matches the exact version you are running.

```
:help              # open the help index
:help {topic}      # search for a specific topic
:help user-manual  # the full narrative user manual (excellent reading)
:help quickref     # a compact quick-reference of all commands
:help tips         # collection of Vim tips from the developers
```

Inside help: `Ctrl+]` follows any hyperlink (shown in `|pipes|`), `Ctrl+O` goes back. Close with `:q`.

vimtutor — the interactive tutorial

If you have not run `vimtutor` yet, do it now. It is a 30-minute hands-on walkthrough that ships with every Vim installation. Unlike reading docs, it teaches you by having you actually edit a file.

```
vimtutor           # run from your terminal — it opens a practice file
```

The tutorial covers the basics thoroughly and is worth revisiting periodically — you will notice things you missed the first time.

Official Documentation Online

vim.org

The official Vim website. Downloads, news, scripts, and links to the documentation. The starting point for anything official.

`official`

Vim User Manual

The complete Vim user manual in HTML — the same content as `:help user-manual` but browsable in a web browser. Excellent reference.

`official docs`

vimhelp.org

A clean, searchable HTML rendering of the full Vim help system. Faster to search than the

Neovim Documentation

The Neovim project's online documentation. Most of it is identical to Vim's — a well-

built-in `:help` when you just want to look something up quickly.

official docs

formatted alternative reference if you are considering the switch to Neovim.

neovim

Vim Tips Wiki

A community wiki with thousands of practical tips, tricks, and how-tos. If you have ever wondered "can Vim do X?", the answer is almost certainly here.

community wiki

r/vim

The Vim subreddit. A friendly and active community — good for questions, discovering plugins, and seeing what other people's setups look like.

community

Cheat Sheets and Quick References

A good cheat sheet on your desk or a second monitor is genuinely useful during the first few months. Here are the best ones available — all free, most printable.

Vim Cheat Sheet — rtorr.com

Widely considered the definitive Vim cheat sheet. Clean layout, covers all modes, motion commands, operators, and more. Available in multiple languages. Print-friendly. This is the one to bookmark.

Vim Cheat Sheet — devhints.io

A well-organised, colour-coded reference card from the devhints collection. Slightly more condensed than rtorr — good for printing on a single page.

VimSheet.com

A focused, no-frills one-page reference. Particularly good for beginners — it prioritises the commands you will actually use rather than listing every single option.

Vim Quick Reference Card (PDF)

A compact two-column PDF designed specifically for printing. Covers motion, editing, search, and Ex commands on a single sheet of paper. By Michael Goerz.

Vim Visual Cheat Sheet — fprintf.net

A keyboard-diagram style reference showing key positions visually. Very useful for understanding the spatial logic of Vim's layout — why the commands are where they are.

Learning Resources

Learning Vim — LinkedIn Learning

The original course by Miki Tebeka that these notes are based on. If you want video walkthroughs of everything covered here, this is where to go.

`video course`

OpenVim

An interactive Vim tutorial that runs in the browser — no installation needed. Good for a quick refresher or for introducing a friend to Vim without asking them to install anything.

`interactive`

Vim Adventures

A browser-based game where you navigate a dungeon using Vim keys. Excellent for building hjkl and motion muscle memory. (Fair warning for left-handers: it was designed for the right-handed keyboard layout!)

`game / practice`

VimGolf

The challenge: transform one piece of text into another in the fewest possible keystrokes. Each puzzle reveals clever solutions from other players. A great way to discover commands you never knew existed.

`game / practice`

The Vim Beginners Site

A curated portal with links to books (most free), tutorials, documentation, and even Vim humour. A good landing page when you want to explore further without knowing exactly what you are looking for.

`portal`

Upcase Vim — thoughtbot

A well-regarded intermediate Vim course from the thoughtbot team. Covers text objects, macros, and navigating large codebases. Free to access.

`video course`

Free Books and Long Reads

Seven Habits of Effective Text Editing

Learn Vimscript the Hard Way

A free online book by Steve Losh covering Vimscript — the language used to write Vim

An essay by Bram Moolenaar, Vim's creator. Not a tutorial — more of a philosophy of efficient editing that explains *why* Vim is designed the way it is. Foundational reading.

`essay / free`

plugins and advanced configuration. The next step after you are comfortable with `.vimrc`.

`book / free`

Practical Vim — Drew Neil

The most-recommended Vim book for intermediate users. Organised as 120 self-contained tips rather than a linear tutorial. You do not read it front to back — you dip in and learn something new each time.

`book / paid`

Vimcasts

A long-running series of short screencasts on Vim topics by Drew Neil (same author as Practical Vim). Over 70 free episodes covering everything from macros to plugins to workflow tips.

`screencasts / free`

Going Beyond Vim: Plugins

Stock Vim is already very capable, but the plugin ecosystem takes it much further. Once you are comfortable with the basics, a small set of well-chosen plugins can close the gap between Vim and a full IDE — while keeping the speed and keyboard-driven workflow that makes Vim worth using in the first place.

Plugin managers

Install a plugin manager first. It handles downloading, updating, and removing plugins through your `.vimrc`.

Manager	Best for	Link
vim-plug	Most users — simple, fast, minimal config	github.com/junegunn/vim-plug
Vundle	Classic option, well documented	github.com/VundleVim/Vundle.vim
lazy.nvim	Neovim users — modern, fast-loading	github.com/folke/lazy.nvim

Plugins worth knowing about

Plugin	What it does
NERDTree	File explorer sidebar — browse and open files with a tree view
fzf.vim	Fuzzy file finder — open any file by typing part of its name
vim-fugitive	Git integration — run git commands, browse diffs, blame, all inside Vim
vim-airline	Enhanced status line showing mode, git branch, file info
ALE	Asynchronous linting — real-time syntax warnings as you type
coc.nvim	Language Server Protocol (LSP) — autocomplete, type hints, go-to-definition
vim-surround	Add, change, or remove surrounding quotes/brackets/tags with a few keys
vim-commentary	Toggle comments on any line or selection with `gcc`
vim-gitgutter	Shows git diff markers (+/−/~) in the sign column as you edit
gruvbox	One of the most popular colour schemes — warm, retro, easy on the eyes

> **Start small with plugins.** The temptation is to install twenty plugins on day one and turn Vim into a mini-IDE. The better path is to learn stock Vim thoroughly first (which you have now done), then add one or two plugins that solve a specific friction point. You will understand what each plugin is doing and why — and your config will stay fast and maintainable.

Should You Switch to Neovim?

Neovim is a fork of Vim started in 2014 with the goal of modernising the codebase. It is fully compatible with Vim — everything in this course works identically in Neovim — but it adds:

- Built-in LSP (Language Server Protocol) support — autocomplete and diagnostics without plugins

- Lua as a first-class configuration language (alongside Vimscript)
- A more active plugin ecosystem with modern tooling
- Better async performance for linting, completion, and search
- Terminal emulator built in

If you are just starting out, do not worry about this. Learn Vim first — every skill transfers. If after a few months you find yourself wanting a richer plugin ecosystem or are doing heavy programming work, Neovim is a natural upgrade path. The official Neovim site has installation instructions and a migration guide.

A Note on Humour

No Vim appendix is complete without acknowledging the memes. Vim has a rich tradition of jokes — mostly about how to quit it (`:q!`, as you now know), the learning curve, and the eternal Vim vs. Emacs debate. If you want to explore the lighter side:

- The Vim Beginners Site has a Vim Humour section worth a browse
- Search "how do I exit Vim" on Stack Overflow — the question has over 4,000 upvotes and the answers have become something of a rite of passage
- VimGolf is genuinely entertaining even when you cannot solve the puzzles — reading other people's solutions teaches you something new every time

> **The best way to get better at Vim** is to stop switching back to another editor when things get difficult. Every time you reach for the mouse, ask: *is there a Vim way to do this?* Usually there is — and once you find it, you will not forget it.