:wq

VIM

Intermediate / Advanced

Macros · Registers In Depth · Advanced Motions & Text Objects

Advanced Search & Substitution · The Plugin Ecosystem

Neovim & Lua Configuration · LSP Integration · A Real Daily Workflow

Philip Osztromok

Generated with Claude

Table of Contents

Vim Intermediate/Advanced — 8 Chapters

CHAPTER	TITLE
Chapter 1	**Macros**
Chapter 2	**Registers In Depth**
Chapter 3	**Advanced Motions & Text Objects**
Chapter 4	**Advanced Search & Substitution**
Chapter 5	**The Vim Plugin Ecosystem**
Chapter 6	**Neovim & Lua Configuration**
Chapter 7	**LSP Integration**
Chapter 8	**Building a Real Daily Workflow**

Chapter 1 — Macros

The Problem Macros Solve

You've likely already used `.` (the dot command) to repeat your last single change — a genuinely useful shortcut, but limited to *one* edit at a time. Real repetitive work is rarely that simple: reformatting fifty lines of a CSV, converting a list of names into function calls, adding the same three-line boilerplate above every function in a file. Each of those is a whole *sequence* of keystrokes you'd otherwise have to repeat by hand, once per line, for however many lines the task touches.

A **macro** is a recorded sequence of any keystrokes — motions, edits, searches, even other commands — stored in a register (Chapter 2) and replayable on demand, as many times as you like. Where the dot command repeats one action, a macro repeats *anything you can type*.

Recording a Macro

`q` followed by a register letter starts recording; pressing `q` again stops it. Everything typed in between — in any mode — becomes part of the macro.

```
qa                # start recording into register 'a'
0i"<Esc>A",<Esc>  # the actual edit: wrap the line in quotes, add a trailing comma
j                 # move down to the next line, ready for the macro's next run
q                 # stop recording
```

> **What gets recorded is literal keystrokes, not intent.** Vim has no idea "you meant to quote the current line" — it only knows you pressed `0`, then `i`, then `"`, then `Esc`, and so on, in that exact order. This is exactly why the `j` at the end of the example above matters: without it, replaying the macro would repeat the edit on the *same* line every time rather than advancing.

Why register a-z, and not 0-9?

Macros are stored in the exact same named registers (Chapter 2) used for yanked and deleted text — there's no separate "macro register" type. Recording into register `a` with `qa...q` and yanking a word into register `a` with `"ayiw` both write to the identical storage slot; whichever happened most recently is what's there now. Pick a register you're not actively using for something else during the same editing session.

Playing Back a Macro

```
@a          # play macro 'a' once
@@          # repeat the LAST macro played — regardless of which register it came from
5@a         # play macro 'a' five times in a row
```

`@@` is worth building into muscle memory early — after playing `@a` once to confirm it worked correctly, `@@` lets you keep repeating it without retyping the register letter each time.

Applying a Macro Across Many Lines

A large count is the simplest way to apply a macro repeatedly down a file:

```
100@a       # attempt to play macro 'a' up to 100 times
```

> **A macro stops naturally when it hits an error.** If macro `a` ends with `j` (move down a line) and you're already on the last line of the file, that `j` fails — and a failed command inside a macro *stops the whole macro* immediately, rather than continuing past it. In practice, this means `100@a` on a 23-line file safely stops after line 23 instead of erroring out or corrupting later lines — you can specify a count far larger than you actually need without worrying about overrunning the file.

Applying a macro to a Visual selection

When you want a macro applied to a specific range rather than "the rest of the file," select the target lines in Visual Line mode first, then invoke the macro via `:normal`:

```
V           # enter Visual Line mode
jjjj        # extend the selection down 4 more lines (5 lines total)
:normal @a  # Vim auto-fills this as :'<,'>normal @a — runs the macro once per selected
line
```

`:normal` executes the given keystrokes as if typed in Normal mode, once for every line in the given range — here, the visually selected range. This sidesteps the "will it stop in time?" question entirely: the macro only ever runs across the lines you explicitly selected.

> **Always test on one line before scaling up.** Whether you're about to run a large count or a `:normal` range, play the macro *once* with a plain `@a` first and visually confirm the result is correct. A macro that silently does the wrong thing (a mistyped keystroke, a search that doesn't match the way you expected) will happily repeat that mistake across every remaining line just as reliably as it would the correct edit — mass-applying a bug is exactly as easy as mass-applying a fix.

Editing a Macro — Since It's Just a Register

Because a macro is stored in an ordinary register, you're not locked into re-recording from scratch if you spot a mistake. You can paste it into the buffer as text, fix the typo, and yank it back:

Workflow — fixing a mistyped macro without re-recording

1	`"ap`	Paste register `a`'s contents into the buffer, on its own line, as plain editable text.
2	`(edit the line)`	The macro's keystrokes are now literal characters — fix the mistyped one directly, same as editing any other text.
3	`"ay$`	Yank from the cursor to the end of the line back into register `a`, overwriting the old (broken) version.
4	`dd (or u)`	Delete the now-unneeded pasted line (or undo the paste) — the fixed macro already lives safely back in register `a`.

Special keys appear as control characters when pasted. An `<Esc>` recorded inside a macro shows up as `^[` when pasted as text (and a literal Enter as `^M`) — these are the actual character, not a two-character sequence, so editing around them works the same as any other single character. It looks unusual the first time you see it, but it pastes and re-yanks correctly regardless.

A Worked Example

Turning a plain list of names into a quoted, comma-terminated list — the kind of transform that comes up constantly when reshaping data for code:

```
# Before:
John Smith
Jane Doe
Alex Chen

# After:
"John Smith",
"Jane Doe",
"Alex Chen",
```

```
# With the cursor on the first line:
qa              # start recording into 'a'
I"<Esc>       # insert a quote at the start of the line
A",<Esc>      # append a closing quote and comma at the end of the line
j               # move to the next line
q               # stop recording

2@a             # apply to the remaining 2 lines (test with @a first, per this chapter's
warn-box, then @@ or a count)
```

Commands Introduced in This Chapter

CHAPTER 1 — COMMAND REFERENCE

RECORDING

`q{register}`	Start recording keystrokes into the given register
`q`	Stop recording (while already recording)

PLAYBACK

`@{register}`	Play the macro stored in the given register, once
`@@`	Repeat the last macro played, regardless of its register
`{count}@{register}`	Play a macro up to `{count}` times, stopping early on the first error

APPLYING ACROSS A RANGE

`V (select lines) then :normal @a`	Run a macro once per line in a Visual selection

EDITING A RECORDED MACRO

`"{register}p`	Paste a macro's contents as editable text
`"{register}y$`	Yank the edited line back into the register as the corrected macro

Chapter 2 — Registers In Depth

What Registers Are For

Every editor has a clipboard — but most editors give you exactly one. Copy something new, and whatever was there before is gone. Vim's answer is **registers**: up to dozens of independent storage slots, each holding its own piece of yanked or deleted text, all available simultaneously.

This matters more than it sounds. Imagine restructuring a function: you need to move three separate blocks of code to three different places. In a single-clipboard editor, you'd yank the first block, paste it, then have to go back and re-select the second block from scratch — because copying it would have overwritten the first. With registers, you can yank all three blocks into three named slots up front, then paste each one exactly where it belongs, in any order.

Chapter 1's macros are themselves stored in a register — this chapter is what makes that fact make sense, and unlocks a much wider set of copy/paste workflows along the way.

The Unnamed Register

Every plain `y` (yank) or `d`/`c` (delete/change) you've used since Chapter 1 of *Learning Vim* already uses a register — you just haven't had to name it. This default is the **unnamed register**, written `""`:

```
yy      # yanks the current line into the unnamed register
dd      # deletes the current line into the unnamed register (overwriting whatever was yanked)
p       # pastes from the unnamed register
```

> **A delete overwrites the unnamed register too.** This is the single most common register surprise: yank a line with `yy`, then delete a different line with `dd` before pasting — the delete silently replaced what you yanked. `p` now pastes the deleted line, not the one you originally copied. Named and numbered registers, below, exist largely to avoid this exact trap.

Named Registers a–z and A–Z

Prefixing any yank, delete, or paste with `"{letter}` targets a specific named register instead of the unnamed one:

```
"ayy      # yank the current line into register 'a'
"bdd      # delete the current line into register 'b'
"ap       # paste from register 'a'
```

The distinction between lowercase and uppercase is deliberate and useful:

Form	Behavior
`"a` (lowercase)	**Overwrites** register `a` — whatever was there before is discarded
`"A` (uppercase)	**Appends** to register `a` — adds to whatever's already there, instead of replacing it

```
# Collect three scattered lines into ONE register, in order:
"ayy      # yank line 1 into 'a' (overwrites)
"Ayy      # yank line 2, APPENDED to 'a'
"Ayy      # yank line 3, APPENDED to 'a'
# register 'a' now holds all three lines, in the order collected
"ap       # paste all three at once
```

> **Appending is how you build a custom clipboard from scattered lines** — walk through a file collecting unrelated lines with `"A`, then paste the whole assembled block in one place. This is a genuinely different capability from a normal clipboard, not just a naming convenience.

Viewing register contents

```
:registers       # list every register and a preview of its contents
:reg             # shorthand for the same command
:reg a b         # list only registers 'a' and 'b'
```

```
--- Registers ---
""   function processOrder(id) {
"0   const result = calculate(x, y)
"a   const BASE_URL = "https://api.example.com"^Jconst TIMEOUT = 5000
"1   old line that was deleted
```

The `^J` you'll sometimes see inside a register's preview represents a newline character — visible proof that register `a` above holds two full lines appended together, not one.

Numbered Registers 0–9

Vim automatically maintains ten numbered registers without you ever naming them explicitly — but they follow two different rules depending on the number.

`"0` — the yank register

Holds the text from your **most recent yank** specifically (`y`, `yy`) — and only a yank. A subsequent delete does *not* touch `"0`, which is exactly what makes it useful: `"0p` reliably pastes "the last thing I actually copied," even after several unrelated deletes.

`"1` – `"9` — the delete ring

Holds a rolling history of your last nine deletions. Each new delete pushes into `"1`, shifting the previous `"1` down to `"2`, `"2` to `"3`, and so on — `"9` falls off the end entirely. `"1p` pastes your most recent delete; `"2p` the one before that.

```
# Demonstrating the split between "0 and the "1-"9 ring:
yy        # yank line A — now in both "" and "0
dd        # delete line B — "" now holds B, "1 holds B, but "0 STILL holds A
"0p       # pastes line A — the yank register was untouched by the delete
"1p       # pastes line B — the most recent delete
```

> **Why this split exists:** deletes happen constantly during normal editing — far more often than deliberate copies. If deletes overwrote the yank register too, `"0` would be useless for anything but the very next keystroke. Separating "things I deleted" from "the thing I deliberately copied" is what makes `"0p` dependable.

Special Registers

A handful of registers are always populated automatically, each recording something specific about your session:

Register	Holds
`"%`	The current file's name
`":`	The most recently executed Ex command (anything typed after `:`)
`"/`	The most recent search pattern
`".`	The last text that was inserted, in Insert mode
`"=`	The **expression register** — evaluates a Vim expression and inserts the result
`"*`	The system clipboard (selection-based, platform dependent)
`"+`	The system clipboard (the one `Ctrl+C` / `Ctrl+V` in other apps actually uses, on most Linux setups)

```
# Insert the current filename into the text itself:
i (enter Insert mode)
Ctrl-r %    # pastes the "% register — the filename — right into the text

# Use "+ to copy INTO the OS clipboard, so it's pasteable in another app:
"+yy      # yank the current line to the system clipboard

# The expression register does math (or any Vim expression) inline:
i (enter Insert mode)
Ctrl-r =12*8<CR>   # inserts "96" directly into the text
```

> `Ctrl-r` **works from Insert mode and the command line alike.** Any register — named, numbered, or special — can be pasted with `Ctrl-r {register}` while typing, without leaving Insert mode to paste and re-enter it. This is the register system's answer to "I need to paste something while I'm already typing."

> `"*` **vs** `"+` **genuinely differs by platform and setup.** On many Linux systems, `"*` reflects the "primary selection" (whatever's currently highlighted, pasted with a middle-click) while `"+` is the clipboard proper (`Ctrl+C` / `Ctrl+V`). On Windows and macOS builds of Vim, the two are often aliased to the same clipboard. If a clipboard yank doesn't paste where you expect in another application, try the other one before assuming clipboard integration is broken.

Practical Workflow: Juggling Multiple Registers

A realistic scenario: refactoring a function's three parameters into a config object, each needing to move to a different destination.

Workflow — moving three separate values to three separate destinations

1	`"ayiw`	Yank the first parameter name (a word, via `iw`) into register `a`.
2	`"byiw`	Move to the second parameter, yank it into register `b`.
3	`"cyiw`	Move to the third, yank it into register `c`. All three are now held independently — none overwrote another.
4	`"ap`	Jump to the config object's first field and paste register `a`.
5	`"bp / "cp`	Repeat for the second and third destinations, in any order — each register still holds exactly what it started with.

Compare this to a single-clipboard workflow, which would require interleaving every yank with its paste — copy one, immediately paste it, go back for the next — because a second copy would destroy the first. Named registers remove that ordering constraint entirely.

Commands Introduced in This Chapter

CHAPTER 2 — COMMAND REFERENCE

TARGETING A REGISTER

`"{register}{operator}`	Direct any yank/delete/change into a specific register
`"{register}p`	Paste from a specific register instead of the unnamed one

NAMED REGISTERS

`"a - "z`	Lowercase — overwrites the register's contents
`"A - "Z`	Uppercase — appends to the register's existing contents

NUMBERED REGISTERS

`"0`	Most recent yank specifically — untouched by deletes
`"1 - "9`	Delete history ring — "1 is most recent, shifting down each new delete

SPECIAL REGISTERS

`"%`	Current filename
`":`	Last Ex command
`"/`	Last search pattern
`".`	Last inserted text
`"=`	Expression register — evaluates a Vim expression
`"* and "+`	System clipboard (selection vs. clipboard proper)

INSERTING A REGISTER WHILE TYPING

`Ctrl-r {register}`	Paste any register directly from Insert mode or the command line

MANAGING REGISTERS

`:registers`	List every register and a preview of its contents
`:reg {letters}`	List only specific registers (e.g. `:reg a b`)

Chapter 3 — Advanced Motions & Text Objects

Beyond Basic Motions

Learning Vim covered motions as ways of moving the cursor — `w` to the next word, `f{char}` to a character, `G` to the end of the file. Combined with an operator (`d`, `c`, `y`), a motion tells Vim *where to stop* — but you still have to think in terms of "where," which usually means precise cursor positioning.

Text objects are a different idea entirely: instead of "where," they describe *what* — "the word," "the paragraph," "the thing inside these parentheses" — regardless of exactly where your cursor happens to sit inside it. This chapter is about that shift, and it's one of the biggest jumps in editing speed once it becomes automatic.

The Text Object Model: Inner vs. Around

Every text object comes in two forms, always following the same pattern: `{operator}{i or a}{object}`.

`i` — inner

Just the content itself — the word, the text between the brackets, the paragraph's lines — with no surrounding delimiter or trailing whitespace included.

`a` — around

The content *plus* its natural surroundings — the delimiter characters themselves, or trailing whitespace/blank line, depending on the object.

```
# cursor anywhere inside "hello":
di"     # delete inner quotes  ->  ""          (quotes remain, content gone)
da"     # delete around quotes ->             (quotes AND content both gone)
```

This `i`/`a` distinction is consistent across every text object below — learn it once, and it applies everywhere.

Word & WORD Text Objects

```
diw     # delete inner word — just the word itself
daw     # delete a word — the word PLUS one side of surrounding whitespace
diW     # delete inner WORD — a whitespace-delimited chunk, ignoring punctuation
boundaries
```

Lowercase `w`ord objects respect punctuation as boundaries (`user_id` and `user-id` are treated differently by `iw`, since `_` counts as a word character but `-` doesn't); uppercase `W`ORD objects only care about whitespace, treating `my-variable.name()` as one single unit. This mirrors the same lowercase/uppercase word-vs-WORD motion distinction from *Learning Vim*'s basic motions — text objects just extend it into operator targets.

> **The cursor doesn't need to be at the start of the word.** Unlike a motion like `dw` (which deletes from the cursor *forward* to the next word boundary), `diw` selects the *whole* word the cursor is currently anywhere inside — including characters to the left of the cursor. Position no longer has to be exact.

Paragraph & Sentence Text Objects

```
dip     # delete inner paragraph — the lines of text, not the blank line after
dap     # delete a paragraph — the lines PLUS the trailing blank line separating it from
the next
dis     # delete inner sentence
das     # delete a sentence, including trailing whitespace
```

A "paragraph," to Vim, is any run of non-blank lines — no special Markdown or prose awareness needed. `dap` is a fast, reliable way to delete an entire block (a function, a config section, a prose paragraph) in one motion, as long as it's separated from its neighbors by a blank line.

Bracket & Quote Text Objects

```
di(  # or di) — inner parentheses
di{  # or di} — inner curly braces
di[  # or di] — inner square brackets
di"  # inner double quotes
di'  # inner single quotes
```

These work from *anywhere* inside the delimiters — you don't need your cursor on the opening or closing character, or even know exactly where they are. This is the single biggest practical upgrade over manual motions like `f(`: no hunting for the matching bracket first.

> **Nested delimiters use the cursor's current level by default.** Inside `outer(middle(inner))`, with the cursor on `inner`, `di(` deletes only `inner` — the innermost pair the cursor is directly inside. To act on the *next level out* instead, prefix with a count: `2di(` deletes `middle(inner)`'s contents, treating the second-innermost pair as the target.

Tag Text Objects (HTML/XML)

```
dit     # delete inner tag — the content between <tag> and </tag>, tags left intact
dat     # delete a tag — the content AND both the opening and closing tags
```

```
# cursor anywhere inside:  <strong>Job Reference:</strong>
citJob Ref<Esc>   # ->  <strong>Job Ref</strong>   (only the text content changed)
```

Tag objects only apply in filetypes where Vim recognizes tag syntax (HTML, XML, and similar) — but for editing markup, `dit` / `dat` / `cit` replace what would otherwise be a fiddly manual selection of exactly the right span of characters.

Combining Operators with Text Objects

Every text object works with every operator — the same small set of building blocks recombines into dozens of practical commands:

Command	Effect
`ciw`	Change inner word — delete the word, drop into Insert mode
`ya{`	Yank around braces — copy the braces and everything inside
`va"`	Visually select around quotes — see the exact span before acting
`dip`	Delete inner paragraph
`>i{`	Indent everything inside braces one level

`v` (Visual mode) combines with text objects too, letting you inspect the exact selection before committing to an operator — useful the first time you try a new object on unfamiliar content.

A Worked Example

Workflow — changing a function argument buried in nested parentheses

1	`/timeout`	Search to land the cursor anywhere near the target — no need to be precise.
2	`ci(`	Change inner parens — deletes everything between the nearest enclosing `(` and `)` and enters Insert mode, regardless of where inside them the cursor landed.
3	`timeout=30<Esc>`	Type the replacement and exit Insert mode — done in three total commands, no manual bracket-matching required.

Compare this to the manual-motion equivalent: locate the opening paren with `f(`, move one right, visually select to the matching close with some combination of `%` and adjustments, then delete. Text objects collapse that whole sequence into one memorable, position-independent command.

Commands Introduced in This Chapter

CHAPTER 3 — COMMAND REFERENCE

THE MODEL

`{operator}i{object}`	Act on the object's content only ("inner")
`{operator}a{object}`	Act on the content plus its surroundings ("around")

WORD OBJECTS

`iw / aw`	Word — respects punctuation boundaries
`iW / aW`	WORD — whitespace-delimited only

PARAGRAPH & SENTENCE OBJECTS

`ip / ap`	Paragraph — a run of non-blank lines
`is / as`	Sentence

DELIMITER OBJECTS

`i( / i{ / i[`	Inner parens/braces/brackets — works from anywhere inside
`i" / i'`	Inner double/single quotes
`{count}i(`	Target an outer level of nested delimiters

TAG OBJECTS

`it / at`	Inner/around an HTML or XML tag

Chapter 4 — Advanced Search & Substitution

Search & Substitute, Beyond the Basics

Learning Vim covered `/pattern` search and a basic `:s/old/new/` substitution. Both go considerably further than that: Vim has its own regex dialect, substitution can target a precise range of lines rather than just "here" or "everywhere," and the global command turns *any* Ex command — not just substitution — into something applied line-by-line across a pattern match.

Vim's Regex Flavor (a Quick Orientation)

If you've worked through the site's own *Learning Regular Expressions* course (`regex1`, built around grep/sed/awk/Bash), be aware: **Vim uses its own regex dialect**, distinct from POSIX BRE/ERE and from PCRE. By default (Vim's "magic" mode), characters like `+` and `?` are treated as *literal* characters, not quantifiers — you'd need to escape them as `\+` and `\?` to get the regex meaning.

```
# default "magic" mode — needs backslashes for + and ?:
:s/colou\?r/color/

# \v "very magic" mode — behaves much closer to PCRE/regex1's conventions:
:s/\vcolou?r/color/
```

> **Start every nontrivial pattern with `\v`.** Prefixing a search or substitution with `\v` switches to "very magic" mode, where most special characters (`+`, `?`, `|`, `(`, `)`, `{`) work the way they do in the regex flavors `regex1` covers, without needing individual backslash-escaping. It's the single easiest way to bring familiar regex intuition into Vim.

Search: New Tricks Beyond /pattern

```
/\cHello      # \c forces case-insensitive, just for this search, regardless of
'ignorecase'
/\<cat\>     # \< and \> are word-boundary anchors — matches "cat", not "category" or
"concat"
```

`gn` is a genuinely powerful combination with Chapter 3's text objects: it selects the *next* match of your last search pattern as a text object in its own right.

```
/TODO<CR>   # search for TODO
cgnNew Task<Esc>   # change the match just found, then...
.                  # ...repeat with the dot command to change the NEXT match too
```

`cgn` followed by repeated `.` presses is often a faster way to change several matches one at a time (with a chance to skip ones you don't want) than a blanket substitution.

The Substitute Command In Depth

```
:s/old/new/      # replaces the FIRST match on the CURRENT line only
:s/old/new/g     # the "g" flag — ALL matches on the current line
:s/old/new/gc    # "g" + "c" — confirm each replacement individually (y/n/a/q)
:s/old/new/gi    # "g" + "i" — case-insensitive for this substitution
```

> **Without the `g` flag, only the first match per line is replaced.** This is the single most common substitution surprise — running `:%s/foo/bar/` across a file where several lines contain `foo` twice will silently leave the second occurrence on each of those lines untouched. If you want every occurrence, the `g` flag isn't optional.

Capture groups from the pattern are available in the replacement as `\1`, `\2`, and so on — and `&` refers to the entire match:

```
:s/\v(\w+)@(\w+)/\2@\1/   # swaps the two halves around an @ symbol
:s/error/[&]/g            # wraps every match in brackets: "error" -> "[error]"
```

Ranges — Applying Substitution to Multiple Lines

Every Ex command, including `:s`, accepts a line range immediately before it:

Range	Applies to
`:5,10s/.../.../g`	Lines 5 through 10
`:%s/.../.../g`	The whole file — `%` is shorthand for `1,$`
`:.,+5s/.../.../g`	The current line through 5 lines below it
`:'<,'>s/.../.../g`	A Visual selection — auto-filled when you press `:` right after selecting

```
# select some lines in Visual mode first:
Vjjjj        # select 5 lines
:              # Vim auto-fills:  :'<,'>
s/foo/bar/g  # type just the command — the range is already there
```

The Global Command :g

`:g/pattern/cmd` runs *any* Ex command against every line matching `pattern` — substitution is only one of many commands it can drive.

```
:g/pattern/d          # DELETE every line matching pattern
:g!/pattern/d         # delete every line NOT matching (or use :v/pattern/d, identical
meaning)
:g/TODO/s/$/ !!/      # append " !!" to the end of every line containing TODO
```

Combined with Chapter 1's macros, `:g` becomes a way to apply a whole recorded sequence to every matching line, not just a substitution:

```
:g/^ERROR/normal @a   # run macro 'a' on every line starting with "ERROR"
```

> **Preview a global command before running it destructively.** `:g/pattern/#` lists every matching line with its line number, without changing anything — a safe way to confirm a pattern matches exactly the lines you intend before following up with `:g/pattern/d` or any other command that actually modifies the file.

A Worked Example

Workflow — cleaning up a config file: strip comments and blank lines

1	`:g/^\s*#/#`	Preview first: list every line that's a comment (starts with optional whitespace then `#`), to confirm the pattern is correct before deleting anything.
2	`:g/^\s*#/d`	Delete every comment line, now that the pattern's been confirmed.
3	`:g/^\s*$/d`	Delete every remaining blank (or whitespace-only) line, leaving only the actual config values.

Commands Introduced in This Chapter

CHAPTER 4 — COMMAND REFERENCE

REGEX MODE	
`\v ...`	"Very magic" mode — closer to standard regex, less escaping needed
SEARCH EXTRAS	
`\c`	Force case-insensitive for this search only
`\< \>`	Word-boundary anchors
`gn`	The next search match, as a text object — combine with operators
SUBSTITUTION FLAGS	
`g`	All matches per line, not just the first
`c`	Confirm each replacement individually
`i`	Case-insensitive for this substitution
`\1, \2, &`	Capture group / whole-match references in the replacement
RANGES	
`:5,10`	A specific line range
`:%`	The whole file
`:'<,'>`	The last Visual selection
THE GLOBAL COMMAND	
`:g/pattern/cmd`	Run any Ex command on every line matching pattern
`:g!/pattern/cmd (or :v)`	Run on every line NOT matching
`:g/pattern/#`	Preview matching lines with line numbers before acting

Chapter 5 — The Vim Plugin Ecosystem

Why Plugins, and Why Carefully

Learning Vim's Configuration chapter covered your `.vimrc` — settings that adjust Vim's existing behavior. Plugins are a different kind of extension entirely: they can add genuinely new commands, new text objects, even new operators, none of which exist in Vim by default. This chapter covers the traditional, vimscript-based plugin ecosystem (usable in both classic Vim and Neovim); Chapter 6 covers Neovim's own Lua-native tooling specifically.

Plugin Managers — vim-plug

vim-plug is one of the most widely used plugin managers — a single vimscript file, installed once, that then handles downloading and loading every plugin you declare:

```
# install vim-plug itself (run once, from a terminal):
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
  https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

Plugins are then declared inside your `.vimrc`, between a matching `plug#begin()` / `plug#end()` pair:

```
" .vimrc
call plug#begin()

Plug 'junegunn/fzf.vim'
Plug 'tpope/vim-surround'
Plug 'tpope/vim-commentary'

call plug#end()
```

```
:PlugInstall   # installs every Plug declared above, not yet on disk
:PlugUpdate    # updates every installed plugin to its latest version
:PlugClean     # removes any installed plugin no longer listed in .vimrc
```

A Curated Set of Essential Plugins

`fzf.vim`

Fuzzy file finding and searching, via `:Files` (jump to any file by a few letters of its name) and `:Rg` (fuzzy-filter live grep results) — genuinely transformative for navigating a large project.

`vim-surround`

Adds commands for adding, changing, and deleting the delimiters *around* text — a direct extension of Chapter 3's text objects, covered in detail below.

`vim-commentary`

Toggle line comments with `gcc` (current line) or `gc` plus any motion — correctly using whatever comment syntax the current filetype expects.

`A file tree (e.g. NERDTree)`

A persistent sidebar file explorer — considerably richer than Vim's built-in `netrw`, closer to the Explorer sidebar from the site's own VS Code course.

vim-surround in depth

This is the plugin most worth understanding well, because it builds directly on Chapter 3's `i` / `a` text-object model rather than introducing an unrelated one:

```
cs"'     # Change Surrounding: double quotes -> single quotes
ds(      # Delete Surrounding: remove the enclosing parentheses, keep the content
ysiw)    # You Surround: wrap the inner word (Chapter 3's iw) in parentheses
```

```
# before:  say(hello)
cs(<          # change surrounding ( to <
# after:   say<hello>
```

> `ysiw)` **reads as a sentence.** `ys` ("you surround") plus `iw` (Chapter 3's inner-word object) plus `)` (the delimiter to add) — "surround the inner word with parentheses." vim-surround deliberately reuses the exact text-object vocabulary from Chapter 3 rather than inventing new motion syntax, which is why it composes so naturally once text objects are already familiar.

Installing and Managing Plugins

Workflow — adding a new plugin to an existing setup

1	`edit .vimrc`	Add a new `Plug '...'` line between the existing `plug#begin()` / `plug#end()` pair.
2	`:source %`	Reload the `.vimrc` you just edited, without restarting Vim.
3	`:PlugInstall`	Installs the newly declared plugin — already-installed ones are left untouched.

> **A plugin runs with your full permissions — evaluate it the way you'd evaluate any dependency.** This is the exact same caution the site's VS Code course gives about extensions: a vimscript plugin isn't sandboxed, and installing one means trusting its author. Before adding a plugin, a quick check of its GitHub stars, last-commit date, and open issues takes seconds and avoids installing something abandoned or, rarely, actively malicious. And just as with VS Code extensions, more isn't better — a handful of plugins genuinely used daily beats a large bundle installed "just in case," each one adding startup time and one more thing to keep updated.

Commands Introduced in This Chapter

CHAPTER 5 — COMMAND REFERENCE

VIM-PLUG

`:PlugInstall`	Install every declared plugin not yet on disk
`:PlugUpdate`	Update all installed plugins
`:PlugClean`	Remove installed plugins no longer declared
`:source %`	Reload the current file (e.g. .vimrc) without restarting

FZF.VIM

`:Files`	Fuzzy-find and open a file by name
`:Rg`	Fuzzy-filter a live grep across the project

VIM-SURROUND

`cs{old}{new}`	Change a surrounding delimiter
`ds{delim}`	Delete a surrounding delimiter, keep the content
`ys{motion/object}{delim}`	Add a surrounding delimiter around a motion or text object

VIM-COMMENTARY

`gcc`	Toggle a comment on the current line
`gc{motion}`	Toggle comments across a motion's range

Chapter 6 — Neovim & Lua Configuration

Neovim vs. Vim — What's Actually Different

Neovim is a fork of Vim, started in 2014 — and everything from *Learning Vim* and this course so far transfers directly: the same modal editing model, the same motions, text objects (Chapter 3), registers (Chapter 2), macros (Chapter 1), and search/substitution (Chapter 4). None of that changes. What differs is the extensibility architecture underneath: Neovim embeds a full Lua runtime alongside vimscript, ships a built-in LSP client (the subject of Chapter 7), an integrated terminal, and asynchronous job control designed in from the start rather than added on later.

> **You don't have to switch to use anything from Chapters 1–5.** Macros, registers, text objects, search/substitution, and vim-plug all work identically in classic Vim. This chapter — and Chapter 7's LSP integration — are the two places this course genuinely requires Neovim specifically, since Neovim's built-in LSP client has no classic-Vim equivalent covered here.

Why Lua?

Vimscript was designed narrowly, for editor configuration — its syntax is quirky by general-programming standards, and historically it's been slow for anything computationally heavier than simple settings. **Lua** is a real general-purpose language, embedded in Neovim via LuaJIT for genuine speed, with syntax many developers already have some familiarity with from other tools. Neovim doesn't just tolerate Lua as an alternative — config, plugin logic, and even keymaps can all be written in it directly, as first-class citizens rather than vimscript wrapped around a function call.

init.lua Basics

Neovim's config file lives at `~/.config/nvim/init.lua`, replacing the classic `.vimrc` / `init.vim` — though either of those still works too (more on that below).

```lua
-- setting options — vim.opt mirrors vimscript's :set
vim.opt.number = true
vim.opt.relativenumber = true
vim.opt.ignorecase = true

-- setting the leader key — the prefix for your own custom mappings
vim.g.mapleader = " "    -- space bar as leader

-- setting a keymap — vim.keymap.set mirrors vimscript's nnoremap
vim.keymap.set('n', '<leader>ff', ':Files<CR>')
```

`vim.keymap.set('n', ...)` — the `'n'` specifies Normal mode, matching the mode-prefix convention from vimscript's own `nnoremap` / `vnoremap` / `inoremap` family. With the leader set to space, `<leader>ff` above means pressing `Space` then `f` then `f` runs `:Files` (Chapter 5's fzf.vim command).

Migrating a vimrc to Lua

vimscript (.vimrc)	Lua (init.lua)
`set number`	`vim.opt.number = true`
`set tabstop=4`	`vim.opt.tabstop = 4`
`set ignorecase`	`vim.opt.ignorecase = true`
`nnoremap <leader>w :w<CR>`	`vim.keymap.set('n', '<leader>w', ':w<CR>')`
`let mapleader = " "`	`vim.g.mapleader = " "`

Migration is optional and gradual, not all-or-nothing. Neovim still reads vimscript directly — an existing `.vimrc` keeps working unchanged, and `init.lua` can even load it explicitly with `vim.cmd('source ~/.vimrc')`. There's no requirement to rewrite everything in Lua at once; many real configs mix both, converting pieces over time as it becomes worthwhile.

Lua-Native Plugin Managers

lazy.nvim is the current standard Lua-native plugin manager (successor to the earlier `packer.nvim`) — plugins are declared as Lua tables, and critically, **lazy-loaded** by default: a plugin's actual code only loads when it's genuinely needed (opening a specific filetype, pressing a specific key), rather than every plugin loading eagerly at startup the way Chapter 5's vim-plug does. This is a real, measurable startup-time advantage as a config grows.

```
-- init.lua, using lazy.nvim
require('lazy').setup({
  { 'tpope/vim-surround' },
  { 'tpope/vim-commentary' },
  {
    'nvim-telescope/telescope.nvim',
    cmd = 'Telescope',    -- only loads when :Telescope is actually run
  },
})
```

A Worked Example — a Minimal init.lua

Workflow — a small but complete init.lua from scratch

1	`vim.g.mapleader = " "`	Set the leader key first — many later keymaps will reference it.
2	`vim.opt.number = true`	Set a handful of basic options, one line per setting.
3	`require('lazy').setup({...})`	Declare plugins in a Lua table, per this chapter's lazy.nvim example.
4	`vim.keymap.set('n', '<leader>ff', ...)`	Add keymaps referencing both the leader key and any newly installed plugin commands.

Commands Introduced in This Chapter

CHAPTER 6 — COMMAND REFERENCE

LUA CONFIG BASICS

`vim.opt.{name} = value`	Set an option — Lua equivalent of :set
`vim.g.mapleader`	Set the leader key
`vim.keymap.set(mode, lhs, rhs)`	Define a keymap — Lua equivalent of nnoremap/vnoremap/etc.
`vim.cmd('...')`	Run a vimscript command from within Lua (e.g. to source an old .vimrc)

LAZY.NVIM

`require('lazy').setup({...})`	Declare and lazy-load plugins from a Lua table

Chapter 7 — LSP Integration

What the Language Server Protocol Is

Before the Language Server Protocol existed, every editor needed its own custom integration for every language it wanted to support well — autocomplete, go-to-definition, inline errors, all built and maintained separately, per editor, per language. With *N* editors and *M* languages, that's *N*×*M* separate integrations.

LSP standardizes the conversation instead: a **language server** (one implementation per language — `pyright` for Python, `gopls` for Go, a TypeScript server for JS/TS) speaks a common JSON-RPC protocol, the same client/server RPC model the site's own `grpc1` course covers from the opposite side. Any editor that implements the *client* half of LSP can talk to *any* language server — *N*+*M* integrations instead of *N*×*M*. Neovim has shipped a built-in LSP client since version 0.5; you don't install a separate plugin for the client itself, only for configuration convenience (below) and completion UI.

nvim-lspconfig — Configuring Language Servers

nvim-lspconfig is a collection of ready-made configuration templates for popular language servers — it is *not* the LSP client itself (that's built in), just the sensible defaults that spare you from hand-writing each server's specific startup command and settings:

```
-- init.lua
require('lspconfig').pyright.setup({})
```

> **Configuring lspconfig doesn't install the language server itself.** `pyright.setup({})` tells Neovim *how* to talk to pyright — but pyright's actual executable still has to exist on your machine first (e.g. `npm install -g pyright`). "I configured lspconfig and nothing happens" is almost always this: the config is correct, but the server binary was never installed.
> **Mason.nvim** is the modern convenience layer that manages installing and updating language server binaries directly from within Neovim, sparing you the separate package-manager step.

Autocompletion with nvim-cmp

The LSP client receives completion *data* from the language server, but Neovim's own built-in completion UI is minimal. **nvim-cmp** is the standard plugin providing an actual completion popup menu, pulling suggestions from multiple sources at once — the LSP client, buffer words, snippets:

```
-- init.lua
local cmp = require('cmp')
cmp.setup({
  sources = {
    { name = 'nvim_lsp' },    -- suggestions from the language server
    { name = 'buffer' },      -- suggestions from words already in the buffer
  },
})
```

Diagnostics — Errors and Warnings Inline

A language server continuously reports **diagnostics** — errors, warnings, hints — which Neovim renders as inline virtual text and gutter signs, updating as you type:

```
]d    # jump to the next diagnostic in the buffer
[d    # jump to the previous diagnostic
:lua vim.diagnostic.open_float()   # show full diagnostic detail for the current line in a floating window
```

This is the same underlying concept as the Problems panel from the site's own VS Code course — errors surfaced directly in the editor rather than only at compile/run time — just rendered inline instead of in a separate panel.

Navigation — Go to Definition, Hover, References

LSP-powered navigation is wired up through Neovim's `vim.lsp.buf` functions, typically bound to keys inside an **on_attach** function — code that runs automatically each time a language server attaches to a buffer:

```
local on_attach = function(client, bufnr)
  vim.keymap.set('n', 'gd', vim.lsp.buf.definition, { buffer = bufnr })
  vim.keymap.set('n', 'K',  vim.lsp.buf.hover,      { buffer = bufnr })
  vim.keymap.set('n', 'gr', vim.lsp.buf.references, { buffer = bufnr })
end

require('lspconfig').pyright.setup({ on_attach = on_attach })
```

Key	Action
`gd`	Go to definition — jump to where the symbol under the cursor is actually defined
`K`	Hover — show documentation/type info for the symbol under the cursor
`gr`	References — list every place the symbol is used across the project

These keymaps reuse the exact `vim.keymap.set` mechanism from Chapter 6, just scoped to the buffer (`{ buffer = bufnr }`) so they only apply where a language server is actually attached, rather than globally.

LSP Is Not the Same as Debugging

It's easy to conflate LSP with the debugging covered in the site's VS Code course (breakpoints, stepping, the call stack) — but they're genuinely different protocols solving different problems. LSP covers *editing-time* intelligence: completion, diagnostics, navigation, all without running your code. Runtime debugging uses a separate standard, the **Debug Adapter Protocol (DAP)** — Neovim's equivalent tooling there is `nvim-dap`, conceptually parallel to VS Code's `launch.json`/breakpoints, but a distinct plugin and protocol from everything in this chapter.

A Worked Example — a Minimal LSP Setup for Python

Workflow — wiring up pyright end to end

1	`npm install -g pyright`	Install the actual language server binary first, per this chapter's own warn-box.
2	`on_attach = function...`	Define the on_attach function with gd/K/gr keymaps, as shown above.
3	`lspconfig.pyright.setup({ on_attach })`	Register pyright with lspconfig, passing in the on_attach function.
4	`cmp.setup({ sources = {...} })`	Wire nvim-cmp to the nvim_lsp source so completions actually appear while typing.
5	`open a .py file`	Diagnostics, gd/K/gr, and completion should all now be active — confirm with K on any function call.

Commands Introduced in This Chapter

CHAPTER 7 — COMMAND REFERENCE

SETUP

`require('lspconfig').{server}.setup({})`	Configure a language server
`require('cmp').setup({...})`	Configure the completion popup and its sources

DIAGNOSTICS

`]d / [d`	Jump to the next/previous diagnostic
`vim.diagnostic.open_float()`	Show full diagnostic detail for the current line

NAVIGATION (VIA ON_ATTACH KEYMAPS)

`gd`	Go to definition
`K`	Hover documentation
`gr`	List references

Chapter 8 — Building a Real Daily Workflow

Bringing It All Together

Seven chapters, seven genuinely separate skills: macros, registers, text objects, search/substitution, plugins, Lua configuration, LSP. Each one is useful in isolation — but a real daily workflow is where they stop being separate topics and start reinforcing each other mid-task, without you consciously switching between "macro mode" and "text object mode." This closing chapter is two realistic scenarios that draw on several chapters at once, a consolidated `init.lua`, and a cheat sheet distilled from the whole course.

Scenario 1 — Refactoring a Data File Into Code

Starting point: a plain-text list of user records, one per line. Goal: a properly formatted array of objects in a source file.

```
# before:
Ada Lovelace, ada@example.com
Grace Hopper, grace@example.com
Alan Turing, alan@example.com

# after:
{ name: "Ada Lovelace", email: "ada@example.com" },
{ name: "Grace Hopper", email: "grace@example.com" },
{ name: "Alan Turing", email: "alan@example.com" },
```

Workflow — combining search/substitute, text objects, and macros

Ch.4	`:%s/\v(.*), (.*)/{ name: "\1", email: "\2" },/`	One global substitution with capture groups handles the bulk of the reformatting across every line at once.
Ch.3	`ci"`	One record has a typo'd email. Land anywhere inside its quotes and change inner quotes directly — no manual selection needed.
Ch.1 / Ch.2	`qa ... q, then "ap`	A later batch of records arrives in a different raw format. Record a macro handling that shape, then paste-and-inspect it (Ch.2's register-editing trick) before trusting it at scale.

Scenario 2 — Navigating and Fixing a Bug Across Files

Workflow — combining plugins, LSP, text objects, and a macro

Ch.5	`:Files`	fzf finds the file mentioned in a bug report by a few letters of its name — no need to know its exact path.
Ch.7	`gd`	Cursor on the suspect function call, jump straight to where it's actually defined via the LSP client.
Ch.3	`ci(`	The bug is a wrong default argument. Change inner parens to fix it, regardless of exactly where inside them the cursor landed.
Ch.7	`gr`	List every reference to this function — several other call sites turn out to have the identical bug.
Ch.1	`qa ci( newvalue <Esc> q, then @a`	Record the fix once as a macro, then replay it at each remaining reference found in the previous step.

Neither scenario needed a "special mode" to move between chapters — the tools composed because they were all designed around the same underlying model from the start (an editable buffer, targetable by motions and text objects, actionable by operators and macros alike).

Assembling a Complete, Minimal init.lua

A consolidated config drawing on Chapters 5–7, small enough to actually read top to bottom:

```lua
-- init.lua
vim.g.mapleader = " "
vim.opt.number = true
vim.opt.ignorecase = true

-- Ch.5/6 — plugins, lazy-loaded
require('lazy').setup({
  { 'tpope/vim-surround' },
  { 'tpope/vim-commentary' },
  { 'junegunn/fzf.vim' },
  { 'neovim/nvim-lspconfig' },
  { 'hrsh7th/nvim-cmp' },
})

-- Ch.7 — LSP, with navigation keymaps
local on_attach = function(client, bufnr)
  vim.keymap.set('n', 'gd', vim.lsp.buf.definition, { buffer = bufnr })
  vim.keymap.set('n', 'K',  vim.lsp.buf.hover,      { buffer = bufnr })
  vim.keymap.set('n', 'gr', vim.lsp.buf.references, { buffer = bufnr })
end
require('lspconfig').pyright.setup({ on_attach = on_attach })

-- Ch.6 — a couple of everyday keymaps
vim.keymap.set('n', '<leader>ff', ':Files<CR>')
vim.keymap.set('n', '<leader>w',  ':w<CR>')
```

Where to Go From Here

`:help` is the single most complete reference for anything this course didn't cover — Vim and Neovim's built-in documentation is genuinely thorough, and `:help {topic}` (e.g. `:help text-objects`) jumps straight to the relevant section.

> **Don't adopt everything from this course at once.** Text objects (Chapter 3) and one or two plugins (Chapter 5) are worth internalizing first — they pay off immediately and rewire how you think about editing. LSP and Lua configuration (Chapters 6–7) are worth adding once that foundation is solid, not on day one. Copying someone else's large, fully-loaded "distro" config wholesale skips the part where you actually understand what each piece does — growing your own `init.lua` a few lines at a time, as a real need comes up, builds configuration you can actually debug later.

Course Cheat Sheet — One Command to Remember Per Chapter

IF YOU ONLY REMEMBER ONE THING PER CHAPTER

CH.1 — MACROS	
`qa ... q, then @a`	Record once, replay as many times as needed
CH.2 — REGISTERS	
`"ayiw / "ap`	Yank into a named register, paste from it — nothing gets silently overwritten
CH.3 — TEXT OBJECTS	
`ci( / di" / dap`	Act on "what," not "where" — works from anywhere inside the object
CH.4 — SEARCH & SUBSTITUTION	
`:%s/\vpattern/replacement/g`	\v for sane regex, g for every match, not just the first per line
CH.5 — PLUGINS	
`ysiw" / cs"' / :Files`	vim-surround extends text objects; fzf makes navigation instant
CH.6 — NEOVIM & LUA	
`vim.keymap.set('n', lhs, rhs)`	The Lua equivalent of nnoremap — config as a real language
CH.7 — LSP	
`gd / K / gr`	Definition, hover, references — install the server binary first
REFERENCE	
`:help {topic}`	The built-in documentation — the real answer to "how do I..."

Course Complete

That's all 8 chapters of **Vim Intermediate/Advanced** — macros, registers, text objects, search & substitution, the plugin ecosystem, Neovim & Lua configuration, LSP integration, and this closing workflow chapter. Combined with the 8 foundational chapters of *Learning Vim*, that's a complete path from first keystrokes to a genuinely modern, LSP-powered editing setup.