

way of the hax0r
n00b to console punk

Chapter 5

Edit Something

Configuration files, scripts, notes, and commit messages are all plain text, and the shell expects you to edit them without leaving the terminal. Sometimes you are on a remote machine with no graphical desktop at all. This chapter gives you two terminal editors, a way to hand a file to a graphical editor from the shell, and the facts about text files that trip people up when they move between systems.

The concepts in this chapter are:

- what a **text editor** does, and how it differs from a word processor;
- **nano**, an editor with its commands on screen;
- **Vim**, a **modal** editor that is installed almost everywhere;
- launching **VS Code** and other graphical editors from the shell;
- the common editing operations and where to find them in each editor;
- **plain text** as bytes plus an **encoding**, and why UTF-8 matters;
- **line endings**, and the difference between LF and CRLF.

The examples create and edit files, so recreate the `hax0r-practice` directory from Chapter 3 and work inside it.

What an Editor Does

A **text editor** loads a file into memory, lets you change that copy, and writes it back when you save. The in-memory copy is called the **buffer**. Until you save, the file on disk is unchanged, which is why quitting without saving discards your edits and why a crash can lose them.

A **word processor** stores fonts, sizes, and layout along with the words. A text editor stores only the characters. That is what programs, configuration files, and the shell itself read, so a document saved from a word processor as `.docx` is not usable where a plain `.txt` or `.sh` is expected.

Every editor offers the same core operations under different keys:

- open a file and move the cursor through it;
- insert and delete text;
- cut, copy, and paste lines;
- search, and search-and-replace;

- undo and redo;
- save, and quit with or without saving.

The differences are in how you reach those operations. nano prints its commands on screen. Vim has **modes**: the keys mean different things depending on whether you are moving around or inserting text. A graphical editor uses menus and the shortcuts you know from other desktop programs.

nano

nano is installed on most Linux systems and on macOS. On Windows it is available inside WSL and Git Bash, not in PowerShell or Command Prompt. [1]

Open a new file:

```
$ nano notes.txt
```

The screen shows the buffer, a status line, and two rows of commands at the bottom. ^ means Ctrl, so ^0 is Ctrl+0, and M- means Alt or Esc followed by the key.

Type a few lines, then:

Action	Keys
Save, called Write Out	Ctrl+O, then Enter to confirm the name
Quit	Ctrl+X; nano asks whether to save unsaved changes
Cut the current line	Ctrl+K
Paste the cut lines	Ctrl+U
Search	Ctrl+W, type the text, Enter
Search and replace	Ctrl+\
Undo and redo	Alt+U, Alt+E
Go to a line number	Ctrl+_
Help	Ctrl+G

Quit with Ctrl+X, answer Y to save, and confirm the filename. Back at the prompt, cat notes.txt prints what you saved. [1]

nano -l shows line numbers, which helps when an error message names one. nano's configuration file is ~/.nanorc; Chapter 10 explains configuration files generally.

Vim

Vim is installed by default on macOS and on most Linux systems, often as vi as well as vim. It's a powerful highly customizable editor that starts quickly with lots of sophisticated commands — perfect for hackers. Learning vi makes many of the other command line utilities feel natural. Git for Windows includes it, and it is the editor Git opens for commit messages unless you configure another. Its keys are different from every other program you use, and the reason is its modes. [2]

Vim starts in **Normal mode**, where letters are commands: j moves down, x deletes a character, and typing a sentence would run a series of commands. Press i to enter **Insert mode**, where typing

inserts text. Press Esc to return to Normal mode. Commands that start with `:` are typed on the bottom line in **Command-line mode** and finished with Enter.

Open a file, insert some text, and save:

```
$ vim notes.txt
```

1. Press `i`. The bottom line shows `-- INSERT --`.
2. Type a line of text.
3. Press Esc.
4. Type `:wq` and press Enter. This writes the file and quits.

Those four steps are enough to edit a commit message or a configuration file. The next table is the survival set; the appendix on Vim keys has more.

Action	Normal-mode keys
Insert before the cursor, or at the end of the line	<code>i</code> , <code>A</code>
Open a new line below and insert	<code>o</code>
Move by character	<code>h</code> , <code>j</code> , <code>k</code> , <code>l</code> , or the arrow keys
Move by word, to line start, to line end	<code>w</code> , <code>b</code> , <code>0</code> , <code>\$</code>
First line, last line, line 42	<code>gg</code> , <code>G</code> , <code>42G</code>
Delete a character, delete a line	<code>x</code> , <code>dd</code>
Copy a line, paste below	<code>yy</code> , <code>p</code>
Undo, redo	<code>u</code> , <code>Ctrl+R</code>
Search forward, next match	<code>/text</code> Enter, <code>n</code>
Save, quit, save and quit	<code>:w</code> , <code>:q</code> , <code>:wq</code>
Quit and discard changes	<code>:q!</code>



Trap: If Vim seems to ignore you, you are probably in the wrong mode. Press Esc twice, then type the command you meant. If the bottom line shows a `:` you did not intend, Esc clears it.

Run `vimtutor` in Bash or zsh for a guided half-hour lesson inside Vim itself. It is the most effective way to learn the basics, and it is already installed with Vim. [2]

Graphical Editors from the Shell

A terminal editor is not the only choice. Launching a graphical editor from the shell keeps the shell's current directory and hands the editor the exact file you mean.

VS Code

`code` opens Visual Studio Code from any of the shells in this book:

```
$ code notes.txt
$ code .
```

The first opens one file; the second opens the current directory as a folder, which is how you usually work on a project. `code --wait notes.txt` keeps the shell waiting until you close the file, which is what Git needs when it uses VS Code for commit messages. [3]

On Windows, the installer adds code to PATH. On macOS, run **Shell Command: Install ‘code’ command in PATH** from the Command Palette once. On Linux, the package installs it. If code is not found, that setup step is missing; Chapter 2 explained how to check PATH.

VS Code’s status bar shows the file’s encoding and line-ending style, and clicking either one changes it. Those two settings are the subject of the next sections.

Other Editors

On Windows, `notepad notes.txt` opens Notepad from PowerShell or Command Prompt. Current Notepad handles UNIX line endings and shows the ending style in its status bar. [4] On macOS, `open -e notes.txt` opens TextEdit, and `open -a "Sublime Text" notes.txt` opens a named application. [5]

Shells and tools such as Git consult the `EDITOR` or `VISUAL` environment variable to decide which editor to launch. Chapter 10 shows how to set it.

Plain Text Is Bytes Plus an Encoding

A file is a sequence of bytes. A **text file** is one where those bytes are meant to be read as characters, and the rule that maps bytes to characters is the **encoding**. The editor and every other program must agree on the encoding, or the characters come out wrong.

- **ASCII** maps the numbers 0 to 127 to English letters, digits, punctuation, and control characters. It has no accented letters and no other scripts.
- **UTF-8** encodes every Unicode character in one to four bytes, and its first 128 values are the same as ASCII. It is the default on Linux and macOS, on the web, and in PowerShell 7.
- **UTF-16** uses two or four bytes per character. Windows uses it internally, and some Windows tools write it to files.
- Older single-byte encodings such as **Windows-1252** or **Latin-1** cover Western European letters and nothing else. They are still the default in some Windows programs. [6]

A **byte order mark**, or **BOM**, is an optional marker at the start of a file that says which of the UTF encodings it uses. UTF-8 does not need one, and many UNIX tools treat a UTF-8 BOM as unexpected characters at the start of the file.

Inspect a file’s encoding before trusting what the editor shows:

Bash/zsh:

```
$ printf 'caf\303\251\n' > utf8.txt
$ file utf8.txt
utf8.txt: Unicode text, UTF-8 text
$ od -c utf8.txt
```

The `printf` writes the two bytes that make up é in UTF-8. `file` guesses the encoding from the bytes, and `od -c` prints every byte, showing the character where it can and the octal value where it cannot.

[7], [8]

iconv converts between encodings:

```
$ iconv -f UTF-8 -t UTF-16 utf8.txt > utf16.txt
$ file utf16.txt
```

PowerShell:

```
$ Set-Content -Path utf8.txt -Value 'café'
$ Get-Content -Path utf8.txt -Encoding utf8
$ Format-Hex -Path utf8.txt
```

Format-Hex shows the bytes. -Encoding on Get-Content and Set-Content chooses the encoding explicitly. [9]



Trap: Windows PowerShell 5.1 and PowerShell 7 have different default encodings. In 5.1, Out-File and > write UTF-16 with a BOM, and Set-Content writes the system's legacy single-byte encoding. PowerShell 7 writes UTF-8 without a BOM everywhere. A file that looks fine in one and garbled in the other was written with the wrong assumption; add -Encoding utf8 to be explicit. [9]

Command Prompt's redirection writes the console's current code page, which is usually not UTF-8. chcp 65001 switches the console to UTF-8 for the session. [10]

Line Endings

A text file marks the end of each line with one or two invisible characters. UNIX-style systems use a single **line feed**, written LF or \n. Windows uses a **carriage return** followed by a line feed, CRLF or \r\n. Both are inherited from typewriters and teleprinters, where returning the carriage and advancing the paper were separate motions. [11]

Most programs read either style. The ones that do not produce confusing symptoms: a shell script that fails with \r: command not found, a Git diff where every line changed, or a file that displays as one long line in an old Windows program.

Make a CRLF file and look at it:

Bash/zsh:

```
$ printf 'one\r\ntwo\r\n' > crlf.txt
$ file crlf.txt
crlf.txt: ASCII text, with CRLF line terminators
$ cat -A crlf.txt
one^M$
two^M$
```

cat -A shows every character: ^M is the carriage return and \$ marks the line feed. Convert to LF by deleting the carriage returns: [12]

```
$ tr -d '\r' < crlf.txt > lf.txt
$ file lf.txt
lf.txt: ASCII text
```

`tr -d` deletes the named character from its input; `<` and `>` connect it to files, which Chapter 7 explains. The `dos2unix` and `unix2dos` tools, where installed, do the same conversion in place. [13]

PowerShell:

```
$ (Get-Content -Raw crlf.txt) -replace "`r`n", "`n" | Set-Content -NoNewline lf.txt
```

`Get-Content -Raw` reads the whole file as one string instead of one string per line, `-replace` swaps the endings, and `-NoNewline` stops `Set-Content` from adding its own ending. [14]

Editors choose an ending for new files based on the operating system, and most keep whatever an existing file uses. VS Code’s status bar and Notepad’s status bar show which one is in effect. Git can convert line endings on the way in and out of a repository, which Chapter 14 covers; decide on one style per project and stick to it.

Key Points

- An editor changes a buffer in memory; the file changes only when you save.
- nano shows its commands on screen: `Ctrl+O` saves, `Ctrl+X` quits.
- Vim is modal: `i` to insert, `Esc` to return to Normal mode, `:wq` to save and quit, `:q!` to abandon.
- `code file` and `code . open` VS Code from the shell; `--wait` makes the shell wait.
- Plain text is bytes plus an encoding. Use UTF-8 unless a tool requires otherwise, and check with `file`, `od`, or `Format-Hex`.
- Windows PowerShell 5.1 and PowerShell 7 default to different encodings; pass `-Encoding` explicitly.
- LF and CRLF are different line endings; `file` and `cat -A` reveal which a file uses, and `tr -d '\r'` converts.

Level Up

n00bs finish the chapter here. You can create and edit a file in nano or Vim, hand a file to VS Code, and diagnose an encoding or line-ending problem. Hackers pick an editor and learn it deeply.

Finish `vimtutor`, then read about Vim’s **text objects** and **counts**: `d3w`, `ci"`, and `>ip`. Explain why Vim users describe editing as a language. Look at **Neovim** and **Helix** and decide what they change about the model.

Try **micro**, a terminal editor with desktop-style shortcuts, and compare it with nano. Try **Emacs** for an afternoon, and notice how many of its keys you already know from Chapter 4.

Investigate **EditorConfig**, a file that tells editors which line ending, indentation, and encoding a project uses. Which of your editors honor it without a plugin?

Open a file in a hex viewer, `xxd` or `Format-Hex`, and find the bytes for a character outside ASCII in your language. Then look up how the same character is encoded in UTF-16 and explain the difference.

Exercises

1. **What does this do?** In Vim's Normal mode, predict the effect of `ddp` on a two-line file, then try it.
2. **What does this do?** Predict the output of `printf 'a\r\nb\n' | cat -A` before running it.
3. **What is the problem?** A script edited on Windows and copied to Linux fails with bad interpreter: `/bin/bash^M`. Explain what the `^M` is and how to fix the file.
4. **What is the problem?** A colleague's PowerShell script writes a list to a file, and `cat` on Linux shows a strange character at the start and spaces between every letter. Name the encoding they probably wrote and the flag that fixes it.
5. **Why is it like this?** Vim starts in Normal mode instead of Insert mode. Explain what that design makes faster, and what it makes harder for beginners.
6. **Why is it like this?** `git commit` opens an editor and waits, but `code notes.txt` returns to the prompt immediately. Explain what `--wait` changes and why Git needs it.
7. **How do you** find out which encoding and line ending a file uses, in Bash and in PowerShell, without opening an editor?
8. **How do you** quit Vim without saving when you have made changes you regret?

References

- [1] Free Software Foundation, “GNU nano: Command-line editor manual.” Available: <https://www.nano-editor.org/dist/latest/nano.html>
- [2] Bram Moolenaar and the Vim Project, “Vim user manual: The first steps in vim.” Available: https://vimhelp.org/usr_02.txt.html
- [3] Microsoft, “Visual studio code: Command line interface.” Available: <https://code.visualstudio.com/docs/editor/command-line>
- [4] Microsoft, “Introducing extended line endings support in notepad.” Available: <https://devblogs.microsoft.com/commandline/extended-eol-in-notepad/>
- [5] Apple, “Open(1): Open files and directories (macOS manual page).” Available: <https://keith.github.io/xcode-man-pages/open.1.html>
- [6] Unicode Consortium, “Unicode FAQ: UTF-8, UTF-16, UTF-32 and BOM.” Available: https://www.unicode.org/faq/utf_bom.html
- [7] Ian F. Darwin and Christos Zoulas, “File(1): Determine file type.” Available: <https://man7.org/linux/man-pages/man1/file.1.html>
- [8] GNU Project, “GNU coreutils manual: Od: Write files in octal or other formats.” Available: https://www.gnu.org/software/coreutils/manual/html_node/od-invocation.html
- [9] Microsoft, “About_character_encoding.” Available: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_character_encoding
- [10] Microsoft, “Windows commands: chcp.” Available: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/chcp>
- [11] Unicode Consortium, “The unicode standard: Newline guidelines (section 5.8).” Available: <https://www.unicode.org/versions/latest/core-spec/chapter-5/#G10213>
- [12] GNU Project, “GNU coreutils manual: Cat: Concatenate and write files.” Available: https://www.gnu.org/software/coreutils/manual/html_node/cat-invocation.html
- [13] GNU Project, “GNU coreutils manual: Tr: Translate, squeeze, and/or delete characters.” Available: https://www.gnu.org/software/coreutils/manual/html_node/tr-invocation.html
- [14] Microsoft, “Get-content.” Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-content>