



Consolidated Reference Notes

A single, de-duplicated reference for the
full laboratory syllabus

LibreOffice Writer

Calc

Impress

Linux Basics

File Structure

Permissions

Accounts & Groups

Vim

Bash

Bash Scripting

Git

L^AT_EX

SSH & PuTTY

Freeplane

How to Use These Notes

The Application Laboratory syllabus is delivered across lecture slides, weekly practical sheets and standalone tutorial handouts. The same material therefore appears several times in slightly different forms: the LibreOffice window anatomy is described three times over (once each for Writer, Calc and Impress), and the core file-and-permission commands recur across the lecture notes, the practical sheets and nearly every assignment.

These notes fold all of that into a single pass. Each concept is explained once, in the place where it first becomes necessary, and later chapters refer back to it rather than repeating it.

The four parts

- **Part I — The LibreOffice Suite.** A shared chapter on the interface elements common to every LibreOffice module, followed by one chapter each for Writer, Calc and Impress.
- **Part II — The Linux Operating System.** What Linux is, how the filesystem is laid out, how to manipulate files and directories, how ownership and permissions work, the Vim editor, and basic system administration.
- **Part III — The Bash Shell.** The shell as an interactive environment (expansion, redirection, pipes, operators), then Bash as a programming language.
- **Part IV — Development and Productivity Tools.** Git, $\text{\LaTeX}$, SSH (including PuTTY on Windows) and Freeplane.

Two appendices close the notes: a consolidated command and shortcut reference for quick lookup during the lab, and a set of pointers for going deeper on each topic after the module ends.

Reading the visual conventions

Convention	Meaning
<code>command</code>	A literal command, function or keyword you type.
<code>/etc/passwd</code>	A file or directory path.
<code>--verbose</code>	A command-line option or flag.
Format → Character	A menu path in a graphical application.
Ctrl+S	A keystroke or key combination.
<i>mount point</i>	A technical term at its first, defining use.

Longer material appears in one of five boxes:

Syntax

States the general form of a command before any example is given. Angle brackets mark values you must supply; square brackets mark optional parts.

Note

Adds context, background or a detail that is easy to overlook but does not change what you type.

Caution

Marks an operation that destroys data, cannot be undone, or fails in a way that is not obvious from the error message.

Tip

A shortcut, an idiom, or a faster way of doing something you already know how to do the long way.

Going further

Points beyond the syllabus: the next concept to learn, the manual page worth reading, or the tool that supersedes what has just been described. Nothing in these boxes is examinable — they exist so the notes stay useful after the module finishes.

Code and terminal blocks

Commands you type are shown with a blue rule on the left:

```
mkdir -p ~/practice/dir1/dir2
cd ~/practice
```

Where the output matters as much as the command, a full session transcript is shown instead, with the shell prompt included:

```
$ ls -l report.txt
-rw-r--r-- 1 student student 0 Aug  4 10:15 report.txt
```

Spreadsheet formulas use a green rule, and $\text{\LaTeX}$ source uses a purple one, so that the three kinds of "code" in this module never get confused with one another:

```
=IF (H2>=350, "Pass", "Fail")
```

```
\section{Introduction}
As shown in Table~\ref{tab:example}, the results are consistent.
```

A note on shell prompts

Transcripts in these notes use `$` for an ordinary user's prompt and `#` for the root user's prompt. Never type the prompt character itself — it is printed by the shell, not by you.

Contents

How to Use These Notes	iii
I The LibreOffice Suite	1
1 The LibreOffice Environment	3
1.1 What LibreOffice is	3
1.2 The Start Center	4
1.3 Anatomy of the window	4
1.4 Styles — the idea behind the whole suite	7
1.5 Templates	8
1.6 Creating, opening and saving	8
2 LibreOffice Writer	11
2.1 What Writer adds	11
2.2 Formatting text	11
2.3 Page layout	13
2.4 Headers and footers	13
2.5 Fields	15
2.6 Tables	15
2.7 Images and drawings	15
2.8 Hyperlinks	16
2.9 Equations	16
2.10 Long-document features	16
3 LibreOffice Calc	19
3.1 What Calc adds	19
3.2 The Calc window	19
3.3 Entering data	21
3.4 Formulas and cell references	21
3.5 Functions	22
3.6 Worked calculation patterns	24
3.7 Formatting	25
3.8 Charts	26
3.9 Opening and saving	26

4	LibreOffice Impress	29
4.1	What Impress adds	29
4.2	Starting Impress	29
4.3	The Impress window	29
4.4	Building a presentation	33
4.5	Changing the look of every slide	34
4.6	Transitions and animations	35
4.7	Running the show	35
II	The Linux Operating System	37
5	Linux Fundamentals	39
5.1	What Linux is	39
5.2	Shell, terminal, console	40
5.3	The anatomy of a command	41
5.4	Getting help	42
5.5	Working efficiently at the prompt	43
5.6	Coming from MS-DOS or Windows	43
5.7	Standards behind the system	44
6	The Linux File Structure	45
6.1	Everything is a file	45
6.2	The single tree	45
6.3	Paths	47
6.4	Hidden files	48
6.5	Naming files	48
6.6	Building a directory hierarchy	49
6.7	Mounting filesystems	49
6.8	How much space	50
7	Working with Files and Directories	53
7.1	Listing — <code>ls</code>	53
7.2	Creating	54
7.3	Copying — <code>cp</code>	55
7.4	Moving and renaming — <code>mv</code>	56
7.5	Deleting	56
7.6	Viewing file contents	58
7.7	Identifying files — <code>file</code>	58
7.8	Wildcards	58

7.9	Navigating between directories	60
7.10	Combining commands on one line	60
7.11	Counting and combining output	61
7.12	A worked structure	61
8	Accounts, Groups and Permissions	63
8.1	Users and groups	63
8.2	The permission model	65
8.3	Changing permissions — <code>chmod</code>	66
8.4	Changing ownership	69
8.5	Default permissions — <code>umask</code>	69
8.6	The special bits	69
8.7	Worked permission changes	70
9	The Vim Editor	73
9.1	Why Vim	73
9.2	Modes — the central idea	74
9.3	Opening, saving and quitting	75
9.4	Moving around	76
9.5	Entering insert mode	77
9.6	The grammar: operator + motion	78
9.7	Editing	79
9.8	Multi-level undo	80
9.9	Visual mode	80
9.10	Registers	81
9.11	Searching and replacing	82
9.12	Macros	82
9.13	Buffers, windows and tabs	83
9.14	Configuring Vim	84
9.15	Plugins	85
9.16	A first session	86
10	Basic System Administration	87
10.1	Shutting down and rebooting	87
10.2	Runlevels and targets	87
10.3	Installing software	89
10.4	Packing and unpacking	89
10.5	Processes	91
10.6	Disk and system information	91

10.7 Networking	92
10.8 Changing your password	92
III The Bash Shell	95
11 The Bash Shell	97
11.1 What the shell is	97
11.2 Variables	97
11.3 Quoting	98
11.4 <code>echo</code> and escape sequences	99
11.5 Expansion	99
11.6 Aliases	101
11.7 Redirection	102
11.8 Pipes	102
11.9 Job control	104
11.10 Debugging what the shell does	104
11.11 History	105
12 Bash Scripting	107
12.1 Anatomy of a script	107
12.2 Variables in scripts	108
12.3 Arguments	108
12.4 Conditionals	109
12.5 Loops	111
12.6 Arrays	112
12.7 Functions	112
12.8 Exit codes	113
12.9 Debugging	113
12.10 Worked scripts	113
12.11 Writing scripts that keep working	116
IV Development and Productivity Tools	117
13 Git	119
13.1 What Git is for	119
13.2 Before you start	119
13.3 The three trees	120
13.4 Your first repository	120
13.5 The everyday cycle	121

13.6	Branching and merging	123
13.7	Working with a remote	124
13.8	Undoing things	125
13.9	Command summary	126
14	LaTeX	127
14.1	What L ^A T _E X is	127
14.2	The compilation cycle	127
14.3	Document anatomy	128
14.4	The title page	129
14.5	Table of contents	130
14.6	Structure	130
14.7	Text formatting	130
14.8	Lists	130
14.9	Mathematics	131
14.10	Tables	132
14.11	Figures	132
14.12	Cross-referencing	133
14.13	Citations and bibliography	133
14.14	Special characters	134
14.15	Common errors	135
14.16	A complete report skeleton	135
15	SSH — Secure Shell	139
15.1	What SSH is	139
15.2	Installing	139
15.3	Connecting	140
15.4	Key-based authentication	141
15.5	Transferring files	142
15.6	The client configuration file	143
15.7	Hardening the server	143
15.8	Troubleshooting	144
15.9	SSH from Windows with PuTTY	144
15.10	Connecting to a virtual machine	146
16	Freeplane and Mind Mapping	149
16.1	What mind mapping is for	149
16.2	Anatomy of a map	150
16.3	The core workflow	150

16.4	Keyboard reference	151
16.5	Formatting for meaning	152
16.6	Structuring a map for reasoning	152
16.7	Saving and exporting	153
V	Appendices	155
A	Command and Shortcut Reference	157
A.1	Navigation and inspection	157
A.2	Files and directories	157
A.3	Wildcards	158
A.4	Permissions and accounts	158
A.5	Shell operators and redirection	159
A.6	Bash scripting	160
A.7	Archiving and packages	161
A.8	System and network	161
A.9	Vim	162
A.10	Git	162
A.11	SSH	164
A.12	LaTeX	164
A.13	LibreOffice keyboard shortcuts	165
A.14	Freeplane	165
B	Going Further	167
B.1	How to keep learning these tools	167
B.2	LibreOffice	167
B.3	Linux and the command line	168
B.4	Vim	168
B.5	Bash	169
B.6	Git	169
B.7	LaTeX	170
B.8	SSH	170
B.9	Freeplane and thinking tools	171
B.10	A suggested order	171

PART I

The LibreOffice Suite

CHAPTER 1

The LibreOffice Environment

Writer, Calc and Impress are three faces of one application. They share a window layout, a toolbar system, a sidebar, a styles engine and a file format. This chapter covers everything the three modules have in common, so that the chapters that follow only have to describe what makes each module different.

1.1 What LibreOffice is

LibreOffice is a free and open-source office suite. It is not a single program but a collection of *modules* that share a common code base, a common user interface and a common document format. The three modules on this syllabus are:

Module	Purpose	Native format	Microsoft equivalent
Writer	Word processing — letters, reports, books	.odt	Word (.docx)
Calc	Spreadsheets — calculation, analysis, charts	.ods	Excel (.xlsx)
Impress	Presentations — slide shows	.odp	PowerPoint (.pptx)

The suite also includes Draw (vector graphics), Base (databases) and Math (formula editing). Math in particular appears indirectly: it is the engine behind the equation objects you insert into a Writer document.

1.1.1 The OpenDocument Format

The native formats above are all *OpenDocument Format* (ODF) files — an open, standardised, XML-based format. An .odt file is in fact a ZIP archive containing XML documents and any embedded images.

This matters for two practical reasons. First, ODF is not owned by any single vendor, so the files remain readable indefinitely. Second, LibreOffice can open and save Microsoft formats, but a round trip through .docx may shift formatting slightly, because the two formats do not describe layout in exactly the same way.

Tip

Work in ODF while you are still editing, and export to .docx or PDF only when you hand the document over. If the recipient does not need to edit it, export to PDF — the layout is then guaranteed to be identical everywhere.

1.2 The Start Center

Opening LibreOffice without specifying a file brings up the *Start Center*. It lists recently used documents as thumbnails, each carrying a small icon showing its type, and offers buttons down the left for creating a new document in each module.

From the Start Center you can:

- click a module button — **Writer Document**, **Calc Spreadsheet**, **Impress Presentation** — to create a blank document;
- click **Open File** to browse for an existing document;
- click **Templates** to start from a pre-designed document.

Once any module is open you no longer need the Start Center: **File** → **New** lets you create a document of any type from inside any module.

1.3 Anatomy of the window

Every LibreOffice module presents the same seven regions. Learning them once means you can find your way around all three.

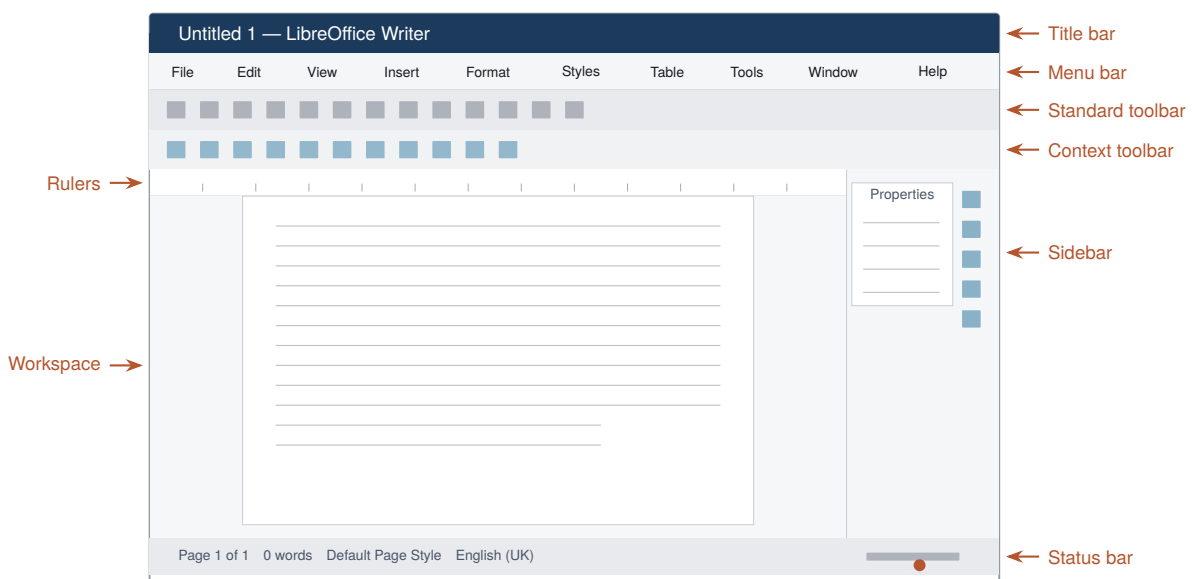


Figure 1.1. The regions common to every LibreOffice module. Only the contents of the workspace and the names on the context toolbar change between Writer, Calc and Impress.

1.3.1 Title bar

Sits at the very top and shows the name of the current document. A newly created document is called **Untitled X**, where **X** is a number, until you save it and give it a name of your own.

1.3.2 Menu bar

Directly under the title bar. Selecting a menu drops down a list of commands. Three visual conventions tell you what a command will do before you click it:

Appearance	Meaning
Plain text	Executes immediately. Save, Close.
Followed by ...	Opens a dialog that asks for more information first. Find and Replace... , Paste Special...
Followed by ►	Opens a further submenu on hover. View → Toolbars, View → Zoom.

The menus common to all modules are **File, Edit, View, Insert, Format, Styles, Tools, Window** and **Help**. Each module then adds its own: Writer adds **Table**, Calc adds **Sheet** and **Data**, Impress adds **Slide** and **Slide Show**.

1.3.3 Toolbars

A toolbar is a strip of icons. LibreOffice displays them in two states:

Docked

Fixed in place along an edge of the window. The default.

Floating

A small free-standing palette that can sit anywhere on screen.

The top docked toolbar, immediately below the menu bar, is the *Standard toolbar*. It is identical in Writer, Calc, Draw and Impress — new, open, save, print, cut, copy, paste, undo, redo and so on.

The second toolbar is *context sensitive*: it changes according to what you have selected. In Writer, working with text shows the Formatting toolbar; click on an image and the Graphics toolbar replaces it, with the Pictures toolbar docking at the bottom of the screen; put the cursor inside a numbered list and a Bullets and Numbering toolbar appears alongside.

Showing and hiding — **View** → **Toolbars**, then click a toolbar name. A tick beside the name means it is currently active.

Moving — To move a docked toolbar, place the pointer over the *toolbar handle* — the small vertical bar at its left end — then hold the left mouse button and drag it to its new position. To move a floating toolbar, drag it by its title bar.

Tear-off toolbars — An icon with a small triangle at its right opens a submenu or a tear-off toolbar. Tear-off toolbars can be dragged away from their parent and then floated or docked like any other. They are not listed under **View** → **Toolbars**.

Customising — Right-click in the space *between* icons on a toolbar to open its context menu. **Visible Buttons** lists every icon defined for that toolbar; ticked entries are shown, and clicking an entry toggles it. The same menu lets you lock a docked toolbar's position.

Single-toolbar mode

If screen space is tight you can collapse the two default toolbars into one. Enable **View** → **Toolbars** → **Standard (Single Mode)** and disable both **Standard** and **Formatting**. The single toolbar carries the most-used commands from both.

1.3.4 Rulers

Horizontal and vertical bars marked with a measurement scale, shown by default along the top and left of the workspace. They show margins as greyed-out regions, and you can drag those boundaries to change the margins directly.

- Show or hide: **View** → **Ruler**.
- Disable permanently: **Tools** → **Options** → **LibreOffice Writer** → **View**, then deselect the rulers you do not want.
- Change the measurement unit: right-click on the ruler and pick from the list. The horizontal and vertical rulers can use different units.

1.3.5 Status bar

Runs along the bottom of the workspace. It reports on the document and doubles as a set of quick controls — most of its fields respond to a single or double click. Hide it with **View** → **Status Bar**.

Several fields appear in all three modules:

Field	Shows / does
Page or slide number	Current position and total count. Double-click to open the Navigator.
Page style	The style applied to the current page or sheet. Double-click to edit it.
Language	The language used for spell checking the selection.
Unsaved changes	An icon indicating whether the document has been modified since the last save. Click it to save.
Digital signature	Present if the document has been signed. Click to view the certificate.
Zoom slider	Drag, or click the + and – ends, to magnify.
Zoom percentage	Click to open the Zoom & View Layout dialog.

The remaining fields are module-specific and are described in the relevant chapters.

1.3.6 Sidebar

The Sidebar is a hybrid of a toolbar and a dialog, docked at the right of the workspace. It is organised into *decks*, each holding several collapsible *panels*.

Action	How
Open a deck	Click its icon in the tab panel at the far right; or click Sidebar Settings at the top right and choose from the list; or use View and pick the deck by name.
Show or hide	View → Sidebar , or Ctrl+F5 , or the Hide/Show marker on the workspace edge.
Close	Click the × at the top right of the Sidebar.
Resize	Drag the Sidebar's left edge.

The decks available depend on the module, but **Properties**, **Styles**, **Gallery** and **Navigator** appear in all three. **Properties** is always context sensitive: its panels change according to whether you have selected text, a cell, an image or a slide.

Tip

Panels in the **Properties** deck carry a **More Options** button that opens the corresponding full dialog. Use the Sidebar for quick adjustments and the dialog when you need the complete set of controls — but note that those dialogs lock the document until you close them.

1.3.7 Tooltips and extended tips

Hovering over an icon, a text box or a menu command shows a *tooltip* — a small box naming the item's function. Move away or press **Esc** to dismiss it.

Extended tips give a longer explanation:

- once only: press **Shift+F1**, then hover;
- from the menu: **Help** → **What's This?**, then hover;
- permanently: **Tools** → **Options** → **LibreOffice** → **General**, and tick **Extended tips**.

1.4 Styles — the idea behind the whole suite

A *style* is a named bundle of formatting. Instead of selecting a heading and manually setting it to 20 pt bold green, you define a style called **Heading 5** once and apply it wherever you need it. Change the style later and every piece of text using it updates at once.

This is the single most important habit to build in LibreOffice, and it is why **Styles** has its own top-level menu. The alternative — selecting text and formatting it by hand — is called *direct formatting*, and it always wins over a style, which is exactly why documents formatted that way become impossible to restyle later.

Style family	Applies to
Paragraph	Whole paragraphs: headings, body text, quotations, captions.
Character	A run of text inside a paragraph: emphasis, code, a defined term.
Page	A whole page: margins, orientation, headers and footers. Writer and Calc.
List	Numbering and bullet schemes, including multi-level outlines.
Frame	Containers for images, text boxes and other floating objects.
Cell	Individual spreadsheet cells. Calc only.
Drawing / Presentation	Shapes and slide elements. Impress and Draw.

Open the styles manager with **F11**, or **Styles** → **Manage Styles**, or **View** → **Styles**. The icons along the top of that window switch between families. The drop-down at the bottom filters which styles are listed — set it to **All** when a style you expect is not showing.

Applying a style — Select the text, then double-click the style in the Styles window — or type its name into the **Set Paragraph Style** box at the left end of the Formatting toolbar and press **Enter**.

Modifying a style — Right-click the style in the Styles window and choose **Modify**. The dialog that opens has one tab per aspect of the formatting — **Font**, **Font Effects**, **Indents & Spacing**, **Borders** and so on.

Creating a style — Format one paragraph exactly as you want it, then click **New Style from Selection** in the Styles window and give it a name.

Organizer and inheritance

Every style dialog has an **Organizer** tab with two important fields. **Inherit from** makes this style borrow everything from a parent style, overriding only what you change — so adjusting the parent cascades down. **Next Style** decides which style the following paragraph gets when you press **Enter**, which is how a heading style can automatically hand over to body text. The same field is what makes alternating left and right page styles possible, as described in Section 2.4.3.

1.5 Templates

A *template* is a document used as a starting point. It carries styles, page setup, and often placeholder content. Creating a document from a template copies all of that into a new, untitled file, leaving the template untouched.

Reach templates through **File** → **New** → **Templates**, or from the Start Center. Save the current document as a template with **File** → **Templates** → **Save as Template**.

1.6 Creating, opening and saving

Creating and opening documents works identically in all three modules.

Command	Use when
Save	Keeping the same file name, location and format. Ctrl+S .
Save As	Creating a new file, or changing the name, the format or the location. Ctrl+Shift+S .
Save Remote	The document lives on, or is going to, a remote server.
Save a Copy	Writing a copy to disk while continuing to edit the original.
Save All	Saving every document open in the current session.

1.6.1 Exporting to PDF

File → **Export as PDF** produces a fixed-layout file that displays identically on any machine. The export dialog is worth exploring: it can embed bookmarks generated from your headings, restrict printing or copying with a password, and control image compression.

Caution

Exporting to PDF is a one-way trip. Keep the ODF original — editing a PDF back into a usable document is far harder than re-exporting from the source.

Going further

Two directions repay the effort once you are comfortable with the basics. *Macros* let you automate repetitive editing; **Tools** → **Macros** supports LibreOffice Basic, Python, BeanShell and JavaScript, and the macro recorder will write a first draft for you. *Base* turns the suite into a database front end and, connected to Writer, drives mail merge from a real data source rather than a spreadsheet. The official user guides at <https://documentation.libreoffice.org> are free, thorough and organised as one book per module.

CHAPTER 2

LibreOffice Writer

Writer is the word processor of the suite. This chapter assumes the shared window layout from Chapter 1 and concentrates on what Writer adds: text formatting, page layout, headers and footers, tables, images and the long-document machinery.

2.1 What Writer adds

Beyond the expected word-processor features — spelling check, thesaurus, hyphenation, auto-correct, find and replace, mail merge — Writer provides:

- templates and styles;
- page-layout tools including frames, columns and tables;
- embedding or linking of graphics, spreadsheets and other objects;
- built-in drawing tools;
- *master documents*, which group a set of shorter documents into one long one;
- change tracking during revision;
- automatic generation of tables of contents and indexes;
- database integration, including a bibliography database;
- export to PDF, with bookmarks.

2.2 Formatting text

Writer separates formatting into two levels. *Character* formatting affects a selected run of text; *paragraph* formatting affects the whole paragraph the cursor sits in, whether or not anything is selected.

2.2.1 Character formatting

Reach the full set through **Format** → **Character**, which opens a dialog with five tabs.

Tab	Controls
Font	Typeface, style (bold, italic), size, language.
Font Effects	Colour, overlining, strikethrough, underlining, relief, outline, shadow, and case transformations.
Position	Superscript and subscript, rotation, scaling, and character spacing (kerning).
Highlighting	Background colour behind the characters.
Borders	A box drawn around the selected characters.
Hyperlink	URL, name, target frame, and the character styles used for visited and unvisited links.

Applying an effect through a style — The preferred route for anything you will use more than once. Select the text, open the Styles window (**F11**), pick the family, right-click the style and choose **Modify**, then set the properties on the **Font** or **Font Effects** tab. For example, redefining **Heading 5** as 20 pt bold green, or **Strong Emphasis** as Georgia 12 pt bold italic dark blue, changes every occurrence in the document at once.

Common effects at a glance —

Effect	Route
Superscript / subscript	Format → Text → Superscript or Subscript . Ctrl+Shift+P / Ctrl+Shift+B .
Overline	Format → Text → Overline .
Strikethrough	Format → Text → Strikethrough .
Shadow	Format → Text → Shadow .
Character spacing	Format → Character → Position tab, then set the amount in the Spacing box.
Highlighting / shading	Format → Character → Highlighting , or the highlight button on the Formatting toolbar.
Rotation and scaling	Format → Character → Position tab.

2.2.2 Paragraph formatting

Format → **Paragraph** controls indentation, spacing above and below, line spacing, alignment, text flow, borders, background — and drop caps.

Drop caps — A *drop cap* is an oversized initial capital that drops into the first few lines of a paragraph. Click inside the paragraph, then **Format** → **Paragraph** → **Drop Caps**, tick **Display drop caps**, and set how many characters to enlarge and how many lines deep they should reach.

Borders — Select the paragraph, then **Format** → **Paragraph** → **Borders**. Set the line arrangement first (which edges get a border), then the style, width and colour.

2.2.3 Bullets and numbering

Simple lists come from the bullet and numbering buttons on the Formatting toolbar. Structured, multi-level lists should come from a *list style*:

1. Open the Styles window and click the **List Styles** icon along the top.
2. Set the filter at the bottom of the window to **All**, so that every available list style is shown.
3. Right-click a style — **Numbering 1**, for example — and choose **Modify**.
4. Go to the **Outline** page and pick the numbering scheme that matches what you need, then adjust it on the **Customize** page.

Tip

Use **Tab** and **Shift+Tab** at the start of a list item to demote and promote it between outline levels. This is far quicker than re-selecting the list style for each level.

2.3 Page layout

Format → **Page Style** opens the page style dialog. Its **Page** tab sets paper format, orientation, margins and layout; the other tabs handle background, borders, columns, footnote area, and the header and footer.

Layout setting	Effect
Right and left	Identical margins on every page.
Mirrored	Inner and outer margins instead of left and right, so that facing pages mirror each other. Use this for anything that will be bound or printed double-sided.
Only right / Only left	Content appears only on right-hand or only on left-hand pages.

Margins can also be dragged directly on the rulers: the grey region at each end of a ruler is the margin, and its boundary is a handle.

2.3.1 Text wrap around objects

Select the object, then **Format** → **Wrap** and choose a mode.

Mode	Result
No Wrap	Text stops above the object and resumes below it.
Page Wrap	Text flows around both sides of the object.
Optimal Page Wrap	Text flows around whichever side has more room.
Wrap Through	Text runs over the object; used with transparent images and watermarks.
In Background	As above, but the object is pushed behind the text.

2.4 Headers and footers

Headers and footers are areas inside the top and bottom page margins that can hold text or graphics. The critical point — and the source of most confusion — is that **a header belongs to a page style, not to a page**. Every page using that style automatically receives the same header.

2.4.1 Adding them

- **Insert** → **Header and Footer** → **Header**, then choose the page style from the submenu. Repeat with **Footer** for the footer.

- Or **Format** → **Page Style**, then the **Header** or **Footer** tab, and tick **Header on** / **Footer on**.

Once the area exists, click into it and type. You can insert fields there — page numbers, chapter headings, the date — as well as plain text.

2.4.2 Formatting them

Direct formatting works on header and footer text just as it does in the body. Beyond that, the **Header** and **Footer** tabs of the page style dialog control the geometry:

- set the spacing between the header and the body text;
- click **More** to open the **Border / Background** dialog and add a border or shadow;
- to draw a separator line between header and body, click the bottom edge of the square in the **Line arrangement** area, then pick a line style;
- to control the gap between that line and the content, clear **Synchronize** and enter a value in the **Bottom** box.

2.4.3 Different headers on different pages

Because a header belongs to a page style, different headers require different page styles. There are two levels of this.

Odd and even pages, same style — The simplest case. On the **Header** tab, clear the **Same content left/right** checkbox. Writer then lets you type a different header on left-hand and right-hand pages within the one style.

Genuinely different page styles — Writer supplies **First Page**, **Left Page** and **Right Page** styles, and you can create your own. Chain them together using the **Next Style** field so that each page automatically hands over to the correct following style:

1. **Format** → **Styles and Formatting**, then click the **Page Styles** icon.
2. Right-click **Right Page** and choose **Modify**.
3. On the **Header** tab, tick **Header on**.
4. Switch to the **Organizer** tab and set **Next Style** to **Left Page**. Click **OK**.
5. Repeat for **Left Page**, setting its **Next Style** back to **Right Page**.
6. Double-click **Right Page** to apply it to the current page.

Now type the header content for the right page; when the document flows onto the next page it picks up **Left Page**, where you type the other header.

Mirrored margins

If the two page styles need different inner and outer margins — as they do in a bound report — set **Page layout** to **Mirrored** on the **Page** tab before defining the headers.

2.4.4 Footnotes

A footnote is not a footer. **Insert** → **Footnote and Endnote** places a reference mark at the cursor and creates the matching note at the bottom of the page; the numbering rennumbers itself

automatically as you edit. The **Footnote** tab of the page style dialog controls how much of the page the footnote area may occupy and what separator line divides it from the body.

2.5 Fields

A *field* is a placeholder that Writer fills in and keeps up to date — page number, total pages, date, time, file name, chapter title, or a cross-reference.

Insert → **Field** offers the common ones directly; **Insert** → **Field** → **More Fields** (**Ctrl+F2**) opens the full dialog. To insert the date and time, use **Insert** → **Field** → **Date** or **Time**.

Note

A date field can be *fixed* (frozen at the moment of insertion) or *variable* (updated every time the document is opened or printed). Choose deliberately — a variable date on a signed letter will silently change.

2.6 Tables

Table → **Insert Table** (**Ctrl+F12**) opens a dialog where you set:

- a **Name** for the table, which is how it appears in the Navigator;
- the number of **Columns** and **Rows**;
- **Heading**, and whether the heading row repeats when the table breaks across a page;
- **Don't split table over pages**, if it must stay whole;
- a **Style** from the AutoFormat list, which sets borders and shading.

With the cursor inside a table, a floating Table toolbar appears with controls for inserting and deleting rows and columns, merging and splitting cells, setting borders and background, and aligning cell contents.

2.7 Images and drawings

2.7.1 Inserting an image

Insert → **Image**, then browse to the file. The dialog has a **Link** checkbox: leave it clear to embed a copy of the image in the document, or tick it to store only a reference to the file on disk. Embedding makes the document self-contained but larger; linking keeps it small but breaks if the image moves.

Right-clicking a placed image gives align, rotate, crop and **Properties** options. Dragging a corner handle resizes it — hold **Shift** while dragging to keep the aspect ratio.

2.7.2 Clip art and the Gallery

Insert → **Media** → **Clip Art** opens the Gallery, which holds themed collections of reusable graphics. Choose an image, then right-click it and select **Insert** to place it in the document.

The Gallery offers two ways to place an object: as a *copy*, which is independent of the original, or as a *link*, which stays tied to it so that changes to the original propagate.

2.7.3 Shapes and grouping

Insert → **Shape** offers lines, basic shapes, symbols, arrows, flowchart elements and callouts. Draw each shape by clicking and dragging.

To combine several shapes so they move and resize as one unit: draw them, select them all (drag a selection rectangle around them, or click each while holding **Shift**), then right-click and choose **Group**. **Ungroup** reverses it.

2.8 Hyperlinks

Two routes, with slightly different capabilities.

Route	Notes
Format → Character → Hyperlink tab	Set the URL, name and target frame, and choose the character styles used for visited and unvisited links. Best when you are attaching a link to text that already exists.
Insert → Hyperlink	Choose the link type first — Internet, Mail, Document, or New Document — then give the path, target and name. Best for links to places inside the same document or to a new file.

2.9 Equations

Insert → **Object** → **Formula** embeds a LibreOffice Math object. The window splits: you type markup in the command panel at the bottom and the typeset result appears in the document. The Elements panel lists the available constructions by category.

```
x = {-b +- sqrt{b^2 - 4 a c}} over {2 a}
```

Click outside the object to return to the document.

Note

Math markup is not $\text{\LaTeX}$, although the two look similar in places. If you need heavy mathematical typesetting, Chapter 14 is the better tool for the job.

2.10 Long-document features

2.10.1 Table of contents

Insert → **Table of Contents and Index** → **Table of Contents, Index or Bibliography**. Writer builds the entries from your *heading paragraph styles*, which is the practical payoff for using styles rather than direct formatting. Right-click the finished table and choose **Update Index** after editing.

2.10.2 Master documents

A *master document* is a container that links several ordinary Writer files as *subdocuments*. Each chapter stays a separate file, but numbering, cross-references, the table of contents and the index are generated across the whole set. Useful for a thesis or a manual where several people edit different chapters.

2.10.3 Change tracking

Edit → **Track Changes** → **Record**. From that point every insertion, deletion and formatting change is marked and attributed. A reviewer can then step through with **Manage** and accept or reject each one individually. **Edit** → **Track Changes** → **Comment** attaches a note to a change.

2.10.4 Mail merge

Tools → **Mail Merge Wizard** combines a Writer document containing database fields with a data source — a spreadsheet, a text file or a Base database — to produce one personalised copy per record.

Going further

The three habits that separate a usable long document from an unmanageable one: use paragraph styles for everything and never format headings by hand; use fields for anything that might change, especially dates and page numbers; and use cross-references rather than typing “see page 14”. All three are the same principle — describe the *role* of a piece of text and let Writer work out its appearance. Chapter 14 takes that principle to its logical conclusion.

CHAPTER 3

LibreOffice Calc

Calc is the spreadsheet module. Its distinctive machinery is the Formula Bar, the cell reference system and the function library — get those three right and everything else follows. This chapter builds up from the interface to a set of worked calculation patterns drawn from the laboratory exercises.

3.1 What Calc adds

You enter data — usually numerical — into a grid, then manipulate it to produce results. Among Calc's features:

- functions, which combine into formulas to perform complex calculations;
- database functions to arrange, store and filter data;
- dynamic charts, in a wide range of 2D and 3D types;
- macros for recording and replaying repetitive tasks, scriptable in LibreOffice Basic, Python, BeanShell and JavaScript;
- the ability to open, edit and save Microsoft Excel spreadsheets;
- import and export in many formats, including HTML, CSV, PDF and PostScript.

3.2 The Calc window

The common regions are as described in Chapter 1. Calc adds the Formula Bar above the workspace, the row and column headers, and the sheet tabs below.

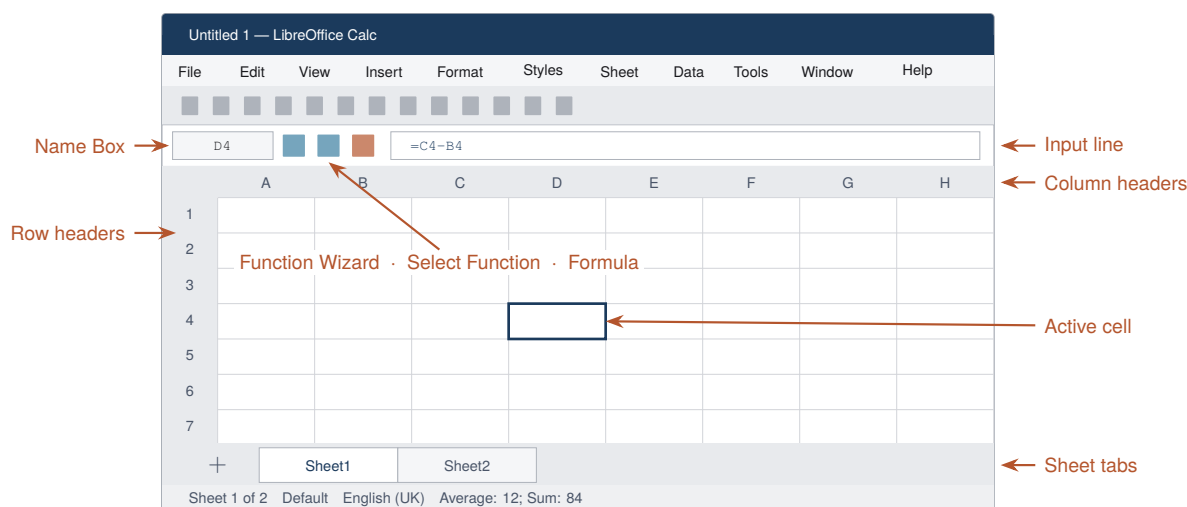


Figure 3.1. Calc's additions to the shared layout: the Formula Bar, the row and column headers that generate cell references, and the sheet tabs.

3.2.1 The Formula Bar

Permanently docked below the toolbars; it cannot float. Show or hide it with **View** → **Formula Bar**. From left to right:

Element	Purpose
Name Box	Shows the reference of the active cell — a column letter plus a row number, such as A1 . If a named range is selected, its name appears instead. You can also type a reference here and press Enter to jump straight to that cell, or type a range name to select the whole range.
Function Wizard	Opens a searchable, categorised list of every available function, showing the argument list and a description for each. Ctrl+F2 .
Select Function	Applies a quick calculation to the numbers above the selected cell — or to the left, if there are none above — and places the result in it. The drop-down offers Sum, Average, Min, Max and Count. Alt+= is equivalent to choosing Sum.
Formula	Inserts an = sign into the cell and the input line, ready for you to type a formula.
Input line	Displays and edits the contents of the selected cell — the underlying data, formula or function, not the displayed result. Click the Expand Formula Bar icon at the right to turn it into a multi-line area for long formulas.

When you start entering data, the **Select Function** and **Formula** icons change into **Cancel** and **Accept**. You can also edit directly inside a cell by double-clicking it.

3.2.2 Cells, rows, columns and sheets

The workspace is a grid. Each *cell* sits at the intersection of a column and a row. Column headers run **A, B, C...** across the top; row headers run **1, 2, 3...** down the side. Together they form the cell reference shown in the Name Box. If the headers are not visible, enable **View** → **Headers**.

A single spreadsheet file can contain many *sheets*, each with its own tab at the bottom of the grid.

Task	How
Create a sheet	Click the + to the left of the sheet tabs.
Switch sheets	Click a tab. The active tab is highlighted.
Select several	Hold Ctrl while clicking tabs.
Rename a sheet	Right-click the tab → Rename Sheet , type the new name, click OK .
Colour a tab	Right-click the tab → Tab Color , pick a colour, click OK .

3.2.3 Calc-specific status bar fields

Beyond the common fields, Calc's status bar reports:

- **Sheet number** — current sheet and total. Note that this is a *sequence* number and may differ from the tab's name if sheets have been moved. Double-click to open the Navigator.

- **Cell selection information** — how many rows and columns are selected. Blank if the selection is not a rectangular block.
- **Insert / overwrite mode** — shows **Overwrite** while editing a cell in overwrite mode, and is blank otherwise. Toggle with the **Insert** key or by clicking the field.
- **Selection mode** — right-click to choose **Standard selection**, **Extending selection** or **Adding selection**.
- **Cell or object information** — with a range selected, this shows the sum of its contents by default. Right-click it to display the average, maximum, minimum or count instead.

3.2.4 Calc's Sidebar decks

Properties carries five panels: **Style**, **Character**, **Number Format**, **Alignment** and **Cell Appearance**. **Styles**, **Gallery** and **Navigator** behave as in the other modules. Calc adds a **Functions** deck, listing every function by category — a simpler alternative to the Function Wizard.

3.3 Entering data

Type into a cell and press **Enter** (move down) or **Tab** (move right). Calc classifies what you type automatically:

Type	Recognised as	Default alignment
1250	Number	Right
12/05/2026	Date	Right
7.5%	Percentage	Right
Saman	Text	Left
=B2+C2	Formula	Result aligned by its type

Autofill

Drag the small square at the bottom-right corner of a selection — the *fill handle* — to extend a series. Calc recognises number sequences, dates, weekdays and month names, and it copies formulas while adjusting their relative references.

3.4 Formulas and cell references

Every formula begins with =. Operators follow ordinary arithmetic precedence, and parentheses override it.

Operators	Meaning
+ - * /	Add, subtract, multiply, divide
^	Exponentiation
%	Percent (divides by 100)
&	Join two pieces of text
= <> < > <= >=	Comparison; these yield TRUE or FALSE
:	Range, as in B4:D14

3.4.1 Relative, absolute and mixed references

This is the single most important concept in spreadsheet work. When you copy a formula, what happens to the references inside it depends on how they are written.

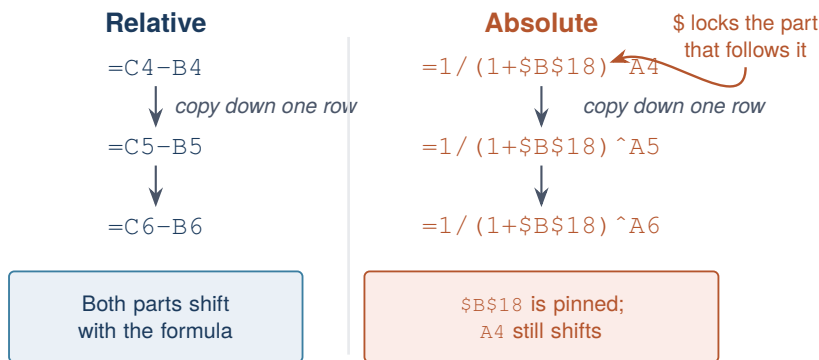


Figure 3.2. Copying a formula rewrites its relative references to match the new position. A \$ freezes the column letter or the row number that follows it.

Form	Name	Behaviour when copied
C5	Relative	Both column and row adjust.
\$C\$12	Absolute	Neither adjusts — always points at C12.
\$C5	Mixed	Column locked, row adjusts.
C\$5	Mixed	Row locked, column adjusts.

Tip

Press **Shift+F4** while the cursor is on a reference in the input line to cycle it through relative, absolute and the two mixed forms.

When to reach for absolute references

Whenever a formula refers to a single fixed input that many rows share — a tax rate, a discount rate, a set of weights. Put that value in one cell, refer to it absolutely, and you can change the assumption in one place and watch the whole model update.

3.5 Functions

A *function* is a named, pre-built calculation. =SUM(B2:B6) is shorter, clearer and less error-prone than =B2+B3+B4+B5+B6.

Find functions through the **Function Wizard** (**Ctrl+F2**), the **Functions** deck in the Sidebar, or **Insert** → **Function**. The wizard is worth using even when you know the name, because it shows the exact argument order.

3.5.1 Core function reference

Function	Returns
<code>SUM (range)</code>	Total of the numbers in the range.
<code>AVERAGE (range)</code>	Arithmetic mean.
<code>MAX (range)</code>	Largest value.
<code>MIN (range)</code>	Smallest value.
<code>COUNT (range)</code>	How many cells hold numbers.
<code>COUNTA (range)</code>	How many cells are non-empty, text included.
<code>ROUND (n; digits)</code>	<code>n</code> rounded to that many decimal places.
<code>IF (test; then; else)</code>	One value if the test is true, another if false.
<code>SUMPRODUCT (r1; r2)</code>	Sum of the pairwise products of two ranges.
<code>NPV (rate; values)</code>	Net present value of a series of future cash flows.
<code>IRR (values)</code>	The discount rate at which net present value is zero.
<code>PMT (rate; nper; pv)</code>	Periodic payment on a loan.
<code>VLOOKUP (key; table; col)</code>	Looks a key up in the first column of a table and returns a value from another column.
<code>CONCATENATE (a; b)</code>	Joins text; <code>&</code> does the same thing.

Argument separators

Calc accepts a semicolon between arguments, and usually a comma as well, depending on your locale settings. These notes use semicolons for functions and commas only where a formula was written that way in the source exercises. If a formula is rejected, swapping the separator is the first thing to try.

3.5.2 Nested IF — assigning bands

An **IF** can be placed inside the *else* slot of another **IF**, producing a chain of tests. This is how a value gets sorted into bands.

Grading, from a final average in column G:

```
=IF (G5>89; "A"; IF (G5>79; "B"; IF (G5>69; "C"; IF (G5>59; "D"; "F"))))
```

The first test runs; if true, "A" is returned and nothing else is evaluated. If false, the next **IF** is evaluated in its place, and so on. Because a false result falls through, you only need to test one boundary at each level.

Tax on taxable income — nil below 250 000, 10% from 250 000 to 500 000, 20% above that:

```
=IF (C2<250000; 0; IF (C2<=500000; C2*0.1; C2*0.2))
```

A remarks column driven by a percentage:

```
=IF (I2>=90; "Excellent"; IF (I2>=80; "Very Good"; IF (I2>=70; "Good"; IF (I2>=60; "Fair"; "Poor"))))
```

Caution

Order matters. Test from one end of the range to the other — highest first, or lowest first — and never mix directions. A chain that tests `G5>59` before `G5>89` awards a D to every student.

3.6 Worked calculation patterns

The exercises in this module reduce to a handful of patterns. Each is given below in the form you would build it.

3.6.1 Summary statistics across a row

Given subject marks in `B2:F2` for each student:

Total	<code>=SUM(B2:F2)</code>
Top subject score	<code>=MAX(B2:F2)</code>
Pass or fail	<code>=IF(G2>=350;"Pass";"Fail")</code>
Percentage	<code>=H2/500*100</code>

Enter each formula once in row 2, then copy it down the column. The relative references follow each student's own row.

3.6.2 Three ways to average a column

All three are equivalent; the third is the one to use.

<code>=(C5+C6+C7+C8)/4</code>	manual, breaks when a row is inserted
<code>=SUM(D5:D8)/4</code>	still hard-codes the count
<code>=AVERAGE(E5:E8)</code>	adjusts by itself

Inserting rows

Insert a row inside a range and Calc widens any range reference that spans it, so `AVERAGE(E5:E8)` becomes `AVERAGE(E5:E9)` automatically. The divide-by-four versions do not: they will produce a wrong answer silently. This is also why a newly inserted row has no formula in its calculated columns — you must copy the formula down into it.

3.6.3 Weighted average with absolute references

Weights in row 12 — `C12` to `F12` — applied to marks in row 5:

relative version	<code>=C5*0.3+D5*0.3+E5*0.3+F5*0.1</code>
absolute version	<code>=\$C\$12*C5+\$D\$12*D5+\$E\$12*E5+\$F\$12*F5</code>

Both give the same answer, but only the second lets you change a weight in one cell and see every student's average update. Copy it down the column: the `$C$12` stays put while `C5` becomes `C6`, `C7` and so on.

3.6.4 Loan EMI

The equated monthly instalment on a loan, given principal `P`, annual interest rate `R` and tenure `N` in years. Lay the three inputs out in their own cells at the top of the sheet and refer to them — never bury a number inside a formula.

```

Monthly interest rate = AnnualRate / 12 / 100
Number of payments   = Years * 12
EMI                   = P*R*(1+R)^N / ((1+R)^N - 1)
Total payment         = EMI * NumberOfPayments
Total interest        = TotalPayment - Principal

```

In the EMI formula, R is the *monthly* rate and N the *total number of monthly payments* — not the annual figures you typed in. Mixing those up is the usual source of a wrong answer.

3.6.5 Discounting and net present value

A cost–benefit model over years 0 to 10, with the discount rate in B18:

```

Total benefit   D4: =C4-B4           copy to D5:D14
Discount factor E4: =1/(1+$B$18)^A4  copy to E5:E14
Present value   F4: =D4*E4           copy to F5:F14
Net present value F16: =SUM(F4:F14)

```

The whole discount-factor column collapses into one expression:

```
=SUMPRODUCT(D4:D14;E4:E14)
```

Or you can skip the intermediate columns entirely with the built-in function. `NPV` assumes each value falls at the *end* of consecutive periods, so a year-0 investment must be added on separately rather than passed in:

```

NPV =NPV(B18;D5:D14)+D4
IRR =IRR(D4:D14)

```

`IRR` returns the discount rate at which the net present value is exactly zero — the break-even cost of capital for the project. Changing B18 and watching the NPV move is the point of building the model this way.

3.7 Formatting

3.7.1 Number formats

Select the range, then **Format** → **Cells** (**Ctrl+1**) and choose a category on the **Numbers** tab.

Category	Notes
Number	Set the decimal places and the thousands separator.
Currency	Pick the symbol and a format such as \$1,234.10; negatives can be shown in red.
Percent	Multiplies the displayed value by 100 and appends %. Set the number of decimal places here.
Date / Time	Many display formats over the same underlying serial number.
Text	Stops Calc interpreting the entry — necessary for identifiers with leading zeros.

Caution

A number format changes only how a value is *displayed*. The stored value is unchanged, so a cell showing 2 may really hold 2.4999 and will calculate as such. If you need the value itself rounded, use `ROUND()`.

3.7.2 Column width and row height

Click the column header letter to select the whole column, then **Format** → **Columns** → **Optimal Width** — or double-click the boundary between two column headers, which does the same thing.

Text that appears to run over into the next cell has not actually moved: the cell to the right is simply empty, so Calc lets the display spill. Widening the column, or turning on wrapping in the **Alignment** panel, fixes it.

3.7.3 Alignment and appearance

The **Alignment** panel of the Sidebar handles horizontal and vertical alignment, text wrapping, indenting, merging cells, text orientation and vertical stacking. **Cell Appearance** handles background colour and borders.

3.8 Charts

Select the data, including its row and column labels, then **Insert** → **Chart**. A four-step wizard follows: chart type, data range, data series, then chart elements (title, axis labels, legend, gridlines).

Double-click a finished chart to re-enter edit mode; a single click selects it as an object for moving and resizing. Charts stay linked to their source data and redraw when it changes.

3.9 Opening and saving

3.9.1 CSV files

A *comma-separated values* file is a plain-text representation of a spreadsheet: one line per row, cell contents separated by a delimiter such as a comma or semicolon, with text usually in quotation marks and numbers without.

1. **File** → **Open** (**Ctrl+O**) and select the file. The extension is usually `.csv`, but some CSV files carry `.txt`.
2. The **Text Import** dialog opens. Set the character set, the separator — comma, semicolon, tab, or something custom — and the text delimiter.
3. Watch the preview at the bottom. If the columns are not splitting where you expect, the separator is wrong.
4. Click **OK**.

Caution

CSV holds data only — no formulas, no formatting, no charts, and only one sheet. Saving a spreadsheet as CSV silently discards all of that. Keep an `.ods` master and export CSV as a delivery format.

3.9.2 Saving

The five save commands behave exactly as described in Chapter 1. Calc's native format is `.ods`; it also reads and writes `.xlsx` and `.xls`.

Going further

Three things extend a spreadsheet from a calculator into an analysis tool. *Named ranges* (**Sheet** → **Named Ranges and Expressions**) let you write `=SUM(Sales)` instead of `=SUM(B4:B120)`. *Pivot tables* (**Data** → **Pivot Table**) summarise large datasets by category without writing a single formula. *Data validation* (**Data** → **Validity**) constrains what a cell will accept, which is how you stop a shared sheet degrading over time. Beyond that, the natural next step is to stop using a spreadsheet as a database and learn SQL.

CHAPTER 4

LibreOffice Impress

Impress is the presentation module. Its structure differs from Writer and Calc in one important way: the workspace has four different views of the same presentation, and knowing which view to be in for a given task is most of the skill.

4.1 What Impress adds

Impress creates presentations in the `.odp` format, and exports to other presentation formats including `.pptx` and PDF. A slide can contain text, bulleted and numbered lists, tables, charts, and graphic objects such as clip art, drawings and photographs. It includes a spelling checker, a thesaurus, text styles and background styles.

4.2 Starting Impress

- From the Start Center: click **Impress Presentation** for a new presentation, or **Open File** to browse for an existing one.
- From the operating system's application menu — the Start menu on Windows, the Applications menu on a Linux Gnome desktop or on macOS.
- From any open LibreOffice module: **File** → **New** → **Presentation**, or the **New** drop-down on the Standard toolbar.

The first time you start Impress, the **Select a Template** dialog appears. The bundled templates are designed for the two standard slide aspect ratios, 4:3 and 16:9, but they adapt to other sizes, which you can pick with **Slide** → **Properties** → **Paper Format** or from the **Slide** section of the **Properties** deck in the Sidebar.

To start with a blank slide instead, click **Cancel** in the template dialog. To stop the dialog appearing at all, go to **Tools** → **Options** → **LibreOffice Impress** → **General** and clear **Start with Template Selection**.

4.3 The Impress window

The main window has three main sections — the Slides pane, the Workspace and the Sidebar — with the menu bar and toolbars above them.

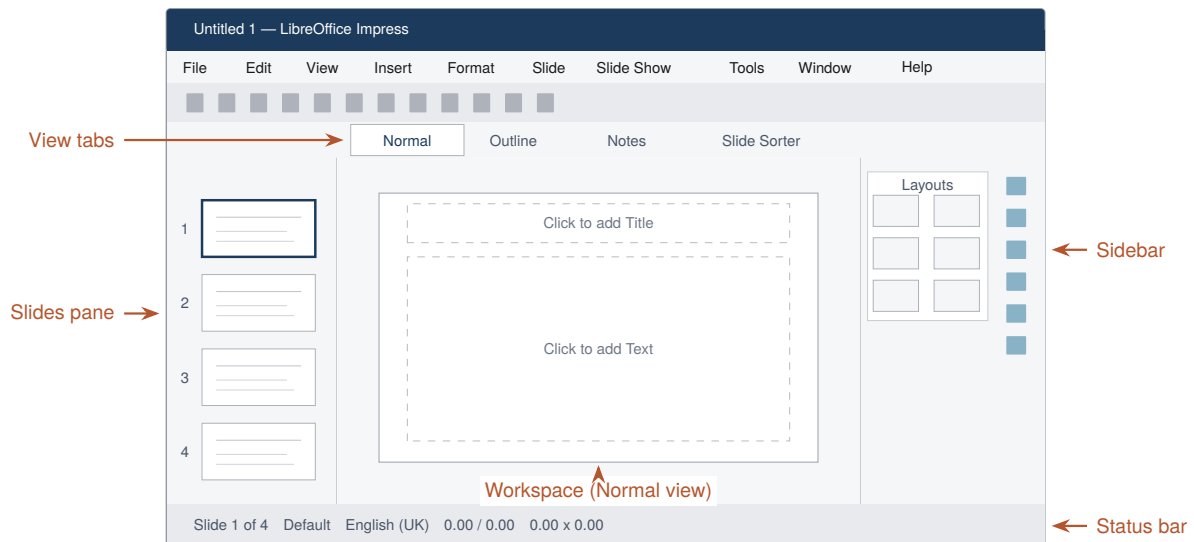


Figure 4.1. Impress in Normal view. The Slides pane on the left, the Workspace in the middle, the Sidebar on the right, and the tabs that switch between the four workspace views.

4.3.1 Slides pane

Holds thumbnails of every slide, in the order they will be shown. Clicking a thumbnail selects that slide and places it in the Workspace for editing.

- Show or close: **View** → **Slides pane**, the × in its top right corner, or the Hide/Show marker on the left of the Workspace.

Right-clicking in the pane gives a context menu that lets you add a new slide, hide a slide so it is skipped during the show, delete, rename or duplicate a slide. Dragging a thumbnail moves the slide to a new position in the order.

4.3.2 Workspace and its four views

The Workspace occupies the centre and opens in Normal view. Four tabs across its top switch between views; if the tabs are missing, enable **View** → **Views Tab Bar**. You can also switch from the **View** menu directly.

View	Use it for
Normal	The main editing view. Design individual slides, add and format text and graphics, apply animation effects. Place a slide here by clicking its thumbnail in the Slides pane or its name in the Navigator.
Outline	Shows every slide's title and bulleted or numbered lists in numbered sequence, as plain outline text. Only text held in the default text boxes appears — text boxes and graphics you added yourself are not shown, and slide names are omitted. Use it to write or restructure the text of the whole deck quickly: edit text, move a paragraph up or down with the arrows on the Outline toolbar, or change its outline level with the left and right arrows. New slides can be created directly here.
Notes	Adds speaker notes to a slide. These are not shown to the audience on an external display. Click the tab, select the slide, then click Click to add notes and type. The text box can be moved and resized by its handles, and text entered into it is formatted with the predefined Notes style, which you can modify in Presentation Styles in the Styles deck.
Slide Sorter	Shows every slide as a thumbnail in a grid. Use it to work on the deck as a whole — reordering, duplicating, hiding or deleting slides.

If you are working with master slides, three further views become available: **Master Slide**, **Master Notes** and **Master Handout**. These have no tabs and can only be reached from the **View** menu.

Working in Slide Sorter view — To change how many slides appear per row: **View** → **Toolbars** → **Slide View** to show the Slide View toolbar, then set the **Slides per Row** box — up to a maximum of 15.

To reorder slides, select one or a group and drag them to the new position. Selecting a group works three ways:

- click the first slide, then hold **Ctrl** and click each of the others — they need not be adjacent;
- click the first slide, then hold **Shift** and click the last, which selects everything between them;
- click and drag a rectangle around the slides you want.

Right-clicking a slide gives **Cut**, **Copy**, **Paste**, **New Slide**, **Duplicate Slide**, **Rename Slide**, **Hide Slide**, **Delete Slide**, **Layout** and **Move**.

4.3.3 Sidebar decks

Impress has seven decks. Four behave as in the other modules; three are specific to presentations.

Deck	Contents
Properties	Nine sections. With a slide selected, Slide and Layouts are available. With an object on a slide selected, you instead get Character , Paragraph , Area , Lists , Shadow , Line , and Position and Size .
Slide Transition	The transitions available, their speed, whether they advance automatically or on a click, and how long a slide is displayed.
Animation	Add, change or remove animations on individual objects, and control how they appear during the show.
Master Slides	The slide design applied to the presentation. The default master is blank; the rest carry backgrounds and styled text.
Styles	Apply, create and modify drawing and presentation styles. Saving a change to a style updates every element formatted with it.
Gallery	Insert an object as a copy (independent of the original) or as a link (which keeps reflecting changes to the original).
Navigator	Move quickly between slides, or select a named object on a slide.

Tip

Give your slides and objects meaningful names rather than leaving them as **Slide 1** and **Shape 1**. The Navigator — opened with **View** → **Navigator** or **Ctrl+Shift+F5** — then becomes genuinely useful for finding things in a long deck.

4.3.4 Rulers and status bar

Rulers appear above and to the left of the Workspace; enable them with **View** → **Rulers**. They show the size of a selected object as a double line, help position object handles and guidelines, and display the page margins as greyed-out regions that can be dragged. Right-click a ruler to change its measurement unit — the two rulers can use different units.

Impress-specific status bar fields:

- **Slide number** — current slide and total.
- **Information area** — changes according to the selected object.
- **Master slide** — the master associated with the current slide. Right-click for a list of available masters; double-click to open the **Available Master Slides** dialog.
- **Cursor position / object size** — with nothing selected, the X and Y coordinates of the pointer. With an object selected, the X and Y of its top-left corner and its width and height. Note that these describe the *selection outline* — the smallest rectangle containing the visible parts of the object — not the object's own geometry. Clicking either area opens the **Position and Size** dialog.
- **Fit slide** — zooms the slide to fill the Workspace.

4.3.5 Toolbars

View → **Toolbars** lists them all. The Standard and Drawing toolbars show by default; **Line and Filling** and **Text Formatting** do not. To change which icons a toolbar shows, right-click an empty area of it and choose **Visible Buttons**.

Toolbar	Purpose
Standard	Identical across all LibreOffice modules.
Drawing	Every function for drawing geometric and freehand shapes and arranging them on a slide.
Line and Filling	Modifies the main properties of a selected object — line colour, style and width, fill colour and style. The icons change with the type of object selected.
Text Formatting	Similar to Writer's Formatting toolbar. Appears automatically when a text object is selected, replacing the Line and Filling toolbar.
Presentation	New Slide, Duplicate Slide, Slide Layout. Not shown by default.

4.4 Building a presentation

4.4.1 Adding slides

Any of these inserts a new slide:

- **Slide** → **New Slide**;
- right-click in the Slides pane → **New Slide**;
- **Ctrl+M**;
- right-click a slide in Slide Sorter view → **New Slide**;
- the **New Slide** icon on the Presentation toolbar.

To duplicate a slide — the copy is inserted immediately after the original — use **Slide** → **Duplicate Slide**, the context menu in the Slides pane or Slide Sorter view, or the **Duplicate Slide** icon on the Presentation toolbar.

4.4.2 Slide layouts

A *layout* determines which content boxes a slide starts with. Impress ships layouts ranging from a blank slide to one with a title and six content boxes.

The first slide is normally a title slide, so **Title Slide** (which includes a subtitle area) or **Title Only** suit it best. For the rest, **Title, Content** is the usual choice.

Select or change a layout in any of four ways: click it in the **Layouts** section of the **Properties** deck; **Slide** → **Layout**; right-click the slide in the Slides pane → **Layout**; or the **Slide Layout** icon on the Presentation toolbar.

Note

Impress cannot create custom layouts. What it can do is let you move, resize and delete the elements of an existing layout, and add new elements anywhere on the slide — which in practice covers the same ground.

4.4.3 Filling content boxes

Each content box can hold one of six things:

Element	How
Title	Click Click to add Title and type. The Text Formatting toolbar opens automatically.
Text	Click Click to add Text and type. Outline styles are applied automatically as you go.
Table	Click Insert Table , give the number of rows and columns, click OK .
Chart	Click Insert Chart . The default chart is placed and the Chart Type section opens in the Sidebar for editing.
Image	Click Insert Image , browse to the file, click Open . The Image section opens in the Sidebar.
Audio or video	Click Insert Audio or Video and browse. The Media Playback section opens in the Sidebar.

4.4.4 Adding text outside a content box

Two routes exist and they behave differently:

Content box

Text typed here picks up the presentation outline styles automatically, and appears in Outline view. Change the outline level of a paragraph with the arrow buttons on the Outline toolbar.

Text box

Click **Insert Text Box** on the Standard or Drawing toolbar, then click on the slide. Type, then click outside to finish. A text box can go anywhere, but its contents do *not* appear in Outline view.

4.4.5 Moving, resizing and removing elements

Task	Steps
Move	Click the outer frame so the selection handles appear. Place the pointer on the frame until it changes shape — usually to a clenched hand. Drag to the new position and release.
Resize	Click the outer frame, then place the pointer on a selection handle until it changes shape and drag. Top and bottom handles change height; left and right change width; corner handles change both.
Remove	Click the box or element to select it, then press Delete or Backspace .

4.5 Changing the look of every slide

To change the background and general appearance of the whole presentation, modify the master slide or select a different one.

A *master slide* is a slide with a defined set of characteristics that acts as a template for the others: background, background objects, text formatting and background graphics. Impress ships with a range of them, listed in the **Master Slides** section of the **Properties** deck. You can create and save your own, or import more.

To change the background of a single slide or of a master:

1. Right-click the slide (or master slide) and choose **Properties** to open the **Slide Setup** dialog.
2. Select **Background**, then pick the type: **None**, **Color**, **Gradient**, **Bitmap**, **Pattern** or **Hatch**.
3. Set the properties offered for that type.
4. Click **OK**.

Tip

Format the master, not the individual slides. Setting the heading font on the master changes every slide at once; setting it slide by slide guarantees that one of them will end up different.

4.6 Transitions and animations

By default a slide show displays every slide in the order shown in Slide Sorter view. Change that order by dragging in Slide Sorter view or in the Slides pane.

Slide transitions — The visual effect *between* slides. Click **Slide Transition** in the Sidebar to choose the transition type, modify it, decide how the slide advances, and preview the result automatically.

Animations — The visual effect applied to an *object on* a slide. Click **Animation** in the Sidebar for the animation category, the specific effect, how it starts, and its direction, duration and delay.

Caution

Both are easy to overdo. A single consistent transition and animation used only where it clarifies — revealing the points of an argument one at a time, for instance — reads as deliberate. A different effect on every slide reads as noise.

4.7 Running the show

Always run the finished presentation through once as a test.

- **Slide Show** → **Start from First Slide**, or **F5**.
- **Slide Show** → **Start from Current Slide**, or **Shift+F5**.

Either opens the Presenter Console automatically. If **Automatically after X seconds** was set, slides advance on their own. If **On mouse click** was set, advance with a mouse click, the arrow keys, the **Spacebar**, or the arrows at the bottom of the Presenter Console. Press **Esc** to end the show and return to Impress.

4.7.1 Presenter Console

Used when a second display is connected. The audience sees only the current slide on the external display, while your own screen shows the current slide, the next slide, your notes for the current slide, and a timer.

Going further

Two habits improve presentations more than any Impress feature. First, build the deck in **Outline** view before you touch a single graphic — if the argument does not hold together as plain text, no amount of design will save it. Second, put your sentences in the **Notes** view and only the headline on the slide; slides carrying full paragraphs compete with you for the audience’s attention and win. When you need a presentation to be reproducible or version-controlled, look at Beamer, the $\text{\LaTeX}$ presentation class — Chapter 14 covers the underlying system.

PART II

The Linux Operating System

Linux Fundamentals

What Linux actually is, why the command line is the way you work with it, and how to read a command. Everything in Parts II and III rests on the grammar described in this chapter.

5.1 What Linux is

Strictly speaking, Linux is a *kernel* — the layer that talks to the hardware, schedules processes, manages memory and mediates access to files. By itself a kernel is unusable.

What people install is a *distribution*: the Linux kernel packaged together with a shell, a set of standard utilities, a package manager, and usually a desktop environment. Ubuntu, Debian, Fedora, Red Hat Enterprise Linux, Arch and Slackware are all distributions of the same kernel with different choices layered on top.

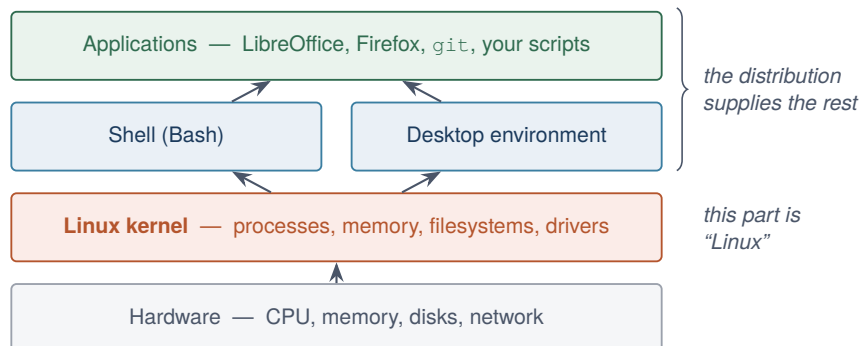


Figure 5.1. Where the kernel sits. The shell is just another program — one whose job is to start other programs on your behalf.

5.1.1 Why Linux is used

Property	What it means in practice
Free and open source	No licence cost, and the source is available for inspection and modification.
One unified tree	Every directory and file appears under a single root, /, rather than being split across drive letters. This makes the system easier to manage and to script.
Better default security	A stricter permission model, and ordinary users do not run with administrative rights.
Extensive customisation	Almost every component — shell, desktop, init system — can be replaced.
Choice of distributions	An enormous number of flavours, each tuned to a different purpose.
Stability	Long uptimes; a misbehaving application is far less likely to bring down the system.
Hardware independence	The kernel has been ported to a very wide range of architectures, from embedded boards to supercomputers.
Lightweight	Will run usefully on hardware that a current Windows release would not.
Consistent updates	The package manager updates the whole system, not just the operating system.
Strong community and documentation	Manual pages ship with the system; forums, wikis and mailing lists are extensive.
Built for development	Compilers, interpreters, editors and version control are all a single command away.
Best choice for servers	It descends from Unix, which was a server operating system to begin with.

On the GUI

Linux has capable desktop environments — GNOME, KDE Plasma, XFCE — and graphical installers. But the command line is where the system is most itself: anything you can do by clicking, you can do by typing, and only what you can type can be scripted, scheduled and repeated exactly. That is why this module teaches the terminal.

5.2 Shell, terminal, console

Three words that are often used interchangeably but mean different things.

Term	Meaning
Shell	The program that reads your commands and runs them. <code>bash</code> is the usual default; <code>sh</code> , <code>zsh</code> , <code>fish</code> and <code>dash</code> are others.
Terminal	The window that displays the shell and passes your keystrokes to it. GNOME Terminal , Konsole and xterm are terminal <i>emulators</i> — they imitate the physical terminals that shells were originally written for.
Console	The text-mode screen the system provides without any graphical environment. Reached with Ctrl+Alt+F3 on most systems.

The prompt printed by the shell tells you who and where you are. A common form is:

```
student@ubuntu:~/documents$
```

— user `student`, on host `ubuntu`, currently in `~/documents`, running as an ordinary user (the `$`). A `#` in place of the `$` means you are root.

Practising without a Linux machine

A browser-based emulator is enough for most of the exercises in this module: <https://copy.sh/v86/?profile=linux26>. For anything involving package installation or SSH, use a real installation or a virtual machine.

5.3 The anatomy of a command

Every command line follows the same grammar.

`ls` `-l` `-a` `/etc`
— — —
command options argument
what to run *how to run it* *what to run it on*

Options begin with `-` (short) or `--` (long). `-l -a` may be combined as `-la`.

Arguments are usually files or directories. Omit them and most commands act on the current directory or on standard input.

Figure 5.2. Command, options, arguments. Whitespace separates them, and everything is case sensitive.

5.3.1 Options

Form	Notes
<code>-l</code>	Short form: a single hyphen and one letter.
<code>-la</code>	Several short options combined after one hyphen.
<code>--all</code>	Long form: two hyphens and a whole word. Easier to read in scripts.
<code>-n 5</code>	An option that takes a value, given after a space.
<code>--lines=5</code>	The same, in long form, joined with <code>=</code> .

Options can change meaning in combination

The letters in an option group are not always independent. `tar` is the classic example: `xvf` and `xvzf` behave differently, because `z` adds gzip decompression to the operation the other letters describe. Read the manual page rather than assuming letters compose freely.

5.3.2 Case sensitivity

Linux is case sensitive throughout — commands, options, file names and directory names.

```
mkdir ABCD
```

```
mkdir abcd
mkdir abCd
mkdir aBCd
mkdir AbcD
```

These create **five separate directories**. On Windows they would all be the same one. This is the single most common source of "but it worked yesterday" errors for people arriving from Windows.

5.4 Getting help

Command	Gives
<code>man ls</code>	The full manual page. Navigate with the arrow keys, search with <code>/pattern</code> , quit with <code>q</code> .
<code>man bash</code>	The Bash manual — long, but the authoritative answer for anything shell-related.
<code>ls --help</code>	A short summary printed straight to the terminal. Faster than <code>man</code> when you just need to check a flag.
<code>info ls</code>	The GNU info manual, often more detailed than the man page.
<code>whatis ls</code>	A one-line description.
<code>apropos copy</code>	Searches the manual page descriptions for a keyword — use it when you do not know the command's name.
<code>which ls</code>	The full path of the program that would run.
<code>type cd</code>	Says whether the name is a program, a shell builtin, an alias or a function.

5.4.1 Reading a manual page

Man pages follow a fixed structure: **NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXAMPLES, SEE ALSO**. In the synopsis, square brackets mark optional parts and italics mark values you supply. Manual pages are grouped into numbered sections, which is why `man 5 passwd` (the file format) and `man 1 passwd` (the command) are different documents.

Tip

`man` is not a last resort. It is faster than a web search for anything about a command's options, it matches the version actually installed on your machine, and it works with no network connection — which matters in a closed laboratory.

5.5 Working efficiently at the prompt

Key	Effect
Tab	Completes a command or path. Press twice to list all possibilities when the completion is ambiguous.
↑ ↓	Step through previous commands.
Ctrl+R	Search backwards through command history incrementally.
Ctrl+A / Ctrl+E	Jump to the start / end of the line.
Ctrl+U / Ctrl+K	Delete to the start / end of the line.
Ctrl+W	Delete the word before the cursor.
Ctrl+L	Clear the screen (same as <code>clear</code>).
Ctrl+C	Interrupt the running command.
Ctrl+D	End of input; at an empty prompt, log out.

Tip

Tab completion is not a convenience, it is an error-prevention tool. If a path does not complete, it does not exist — you have caught a typo before running a command that might have created something in the wrong place.

5.6 Coming from MS-DOS or Windows

The concepts transfer; the names do not.

MS-DOS / Windows	Linux	Purpose
<code>md / mkdir</code>	<code>mkdir</code>	Create a directory
<code>rd</code>	<code>rmdir</code>	Remove an empty directory
<code>dir</code>	<code>ls</code>	List directory contents
<code>del</code>	<code>rm</code>	Delete a file
<code>copy</code>	<code>cp</code>	Copy a file
<code>move / ren</code>	<code>mv</code>	Move or rename
<code>type</code>	<code>cat</code>	Show a file's contents
<code>cls</code>	<code>clear</code>	Clear the screen
<code>cd</code>	<code>cd</code>	Change directory
<code>ipconfig</code>	<code>ip a / ifconfig</code>	Show network interfaces
<code>ping</code>	<code>ping</code>	Test reachability of a host
<code>tracert</code>	<code>traceroute</code>	Trace the route to a host
<code>\</code>	<code>/</code>	Path separator
<code>C: D:</code>	<code>/</code> and mount points	Storage is addressed

Browsers on Linux

Firefox, Google Chrome, Chromium, Brave and Opera all run natively on Ubuntu. Microsoft Edge has a Linux build. Internet Explorer never did and never will — it was discontinued, and was always tied to Windows components.

5.7 Standards behind the system

Several standards keep distributions broadly compatible with each other:

POSIX

The Portable Operating System Interface — a family of IEEE standards defining the system calls, the shell language and the standard utilities. Writing to POSIX means a script will run on Linux, macOS and the BSDs alike.

Single UNIX Specification (SUS)

The specification a system must meet to be certified and marketed as UNIX.

Filesystem Hierarchy Standard (FHS)

Defines what belongs in each top-level directory. It is why `/etc` means the same thing on Ubuntu and on Fedora. Covered in Chapter 6.

Linux Standard Base (LSB)

A joint project of several distributions under the Linux Foundation to standardise the software system structure, including the FHS. It was based on POSIX, the Single UNIX Specification and several other open standards, extending them in certain areas. The project is no longer actively developed, but much of what it standardised is now simply how things are done.

Going further

Two things are worth doing early. Install a Linux virtual machine — VirtualBox with Ubuntu Server is free and gives you a system you can break safely, which also gives you somewhere real to practise SSH (Chapter 15). And read `man hier`, a single manual page describing the whole filesystem layout — it takes ten minutes and makes Chapter 6 obvious. For a book, *The Linux Command Line* by William Shotts is freely available at <https://linuxcommand.org/tlcl.php> and covers Parts II and III of these notes in far greater depth.

CHAPTER 6

The Linux File Structure

Linux presents all storage as one tree. This chapter covers that tree, what lives where, how to describe a location in it, and how additional storage is grafted on by mounting.

6.1 Everything is a file

In Linux, almost everything is represented as a file: ordinary documents, of course, but also directories, devices, running-system information and network sockets. Reading a keyboard and reading a text file use the same system calls. This is why the same handful of commands — `cat`, `ls`, `cp` — work on such a wide range of things.

The first character of a long listing tells you which kind of object you are looking at.

Char	Type	Meaning
-	Regular file	Text or binary data.
d	Directory	A file whose contents are a list of other files.
l	Symbolic link	A pointer to another path.
c	Character special	A device transferring data character by character — a terminal, a serial port.
b	Block special	A device transferring data in fixed-size blocks — a disk.
s	Socket	An endpoint for network or inter-process communication.
p	Named pipe (FIFO)	A pipe with a name in the filesystem.

```
$ ls -l
drwxr-xr-x  2 root root 4096 Apr 14 03:15 folder1
-rw-r--r--  1 root root   0 Apr 14 03:29 samplecopy.txt
lrwxrwxrwx  1 root root   9 Apr 14 03:31 shortcut -> /etc/hosts
```

6.2 The single tree

There are no drive letters. Every file on every storage device appears somewhere beneath a single root directory, written `/`.

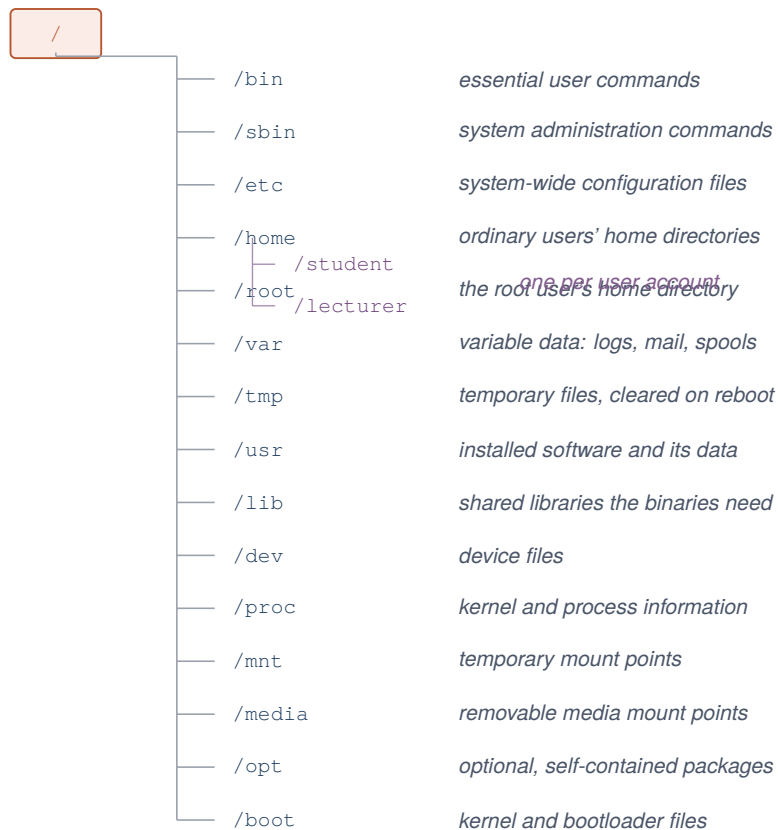


Figure 6.1. The main top-level directories. The layout is defined by the Filesystem Hierarchy Standard, which is why it is the same across distributions.

6.2.1 The directories you will actually use

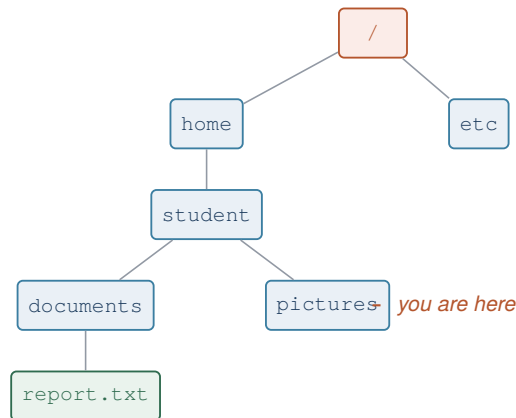
Path	Notes
<code>/home/user</code>	Your own files. Writable by you, and normally by nobody else. Abbreviated <code>~</code> in the shell.
<code>/root</code>	The root user's home directory. Note that this is <i>not</i> <code>/</code> , despite the name.
<code>/etc</code>	Configuration files, in plain text. <code>/etc/passwd</code> , <code>/etc/group</code> , <code>/etc/ssh/sshd_config</code> , <code>/etc/hosts</code> . Editing these usually requires root.
<code>/bin</code> , <code>/usr/bin</code>	Executable programs available to all users — <code>ls</code> , <code>cp</code> , <code>cat</code> .
<code>/sbin</code> , <code>/usr/sbin</code>	Programs intended for the system administrator — <code>fdisk</code> , <code>useradd</code> .
<code>/var</code>	Data that changes as the system runs. <code>/var/log</code> is where you go when something has gone wrong.
<code>/tmp</code>	Scratch space. World-writable, and cleared on reboot — never leave anything here that matters.
<code>/dev</code>	Device files. <code>/dev/sda</code> is a disk; <code>/dev/null</code> discards everything written to it.
<code>/proc</code>	A virtual filesystem exposing kernel and process state as files. <code>cat /proc/cpuinfo</code> reads your processor details.
<code>/mnt</code> , <code>/media</code>	Where other filesystems get attached.

Note

On modern distributions `/bin`, `/sbin` and `/lib` are symbolic links into `/usr`. The distinction is historical — it dates from when `/usr` might live on a separate disk that was not available at boot.

6.3 Paths

A *path* is the address of a file or directory. There are two ways to write one, and choosing correctly is most of what "I can't find my file" comes down to.



Absolute path — starts at `/`, works from anywhere

`/home/student/documents/report.txt`

Relative path — starts from the current directory

`../documents/report.txt`

Figure 6.2. The same file, described two ways, from the `pictures` directory.

Kind	Rule
Absolute	Begins with <code>/</code> . Traces the complete route from the root of the tree. Means the same thing no matter where you are standing.
Relative	Does <i>not</i> begin with <code>/</code> . Interpreted starting from your current working directory — so it means different things in different places.

6.3.1 Path shortcuts

Symbol	Means
<code>.</code>	The current directory. <code>cp file.txt .</code> copies into where you are.
<code>..</code>	The parent directory. <code>cd ..</code> moves up one level; <code>ls -l ..</code> lists the parent without moving there.
<code>~</code>	Your own home directory. <code>~/notes</code> is <code>/home/student/notes</code> when you are <code>student</code> .
<code>~bob</code>	Another user's home directory.
<code>-</code>	The directory you were in previously. <code>cd -</code> toggles between two directories.
<code>/</code>	The root of the tree, when it appears at the start of a path.

Tip

Two rules that resolve most path confusion. Use *absolute* paths in scripts and in anything scheduled — the working directory at run time is not under your control. Use *relative* paths interactively, where they are shorter and you can see where you are.

6.3.2 Knowing where you are

```
pwd          # print working directory
cd /etc      # absolute: go to /etc
cd ..        # relative: up one level
cd ~/documents # into documents, inside your home
cd           # with no argument: straight home
cd -         # back to the previous directory
```

Note

Because `cd` with no argument goes home, and `~` means home, you can operate on your own files from anywhere without navigating first: `mkdir ~/newdir` creates a directory in your home even while you are standing in `/etc`.

6.4 Hidden files

A file whose name begins with a dot is hidden from ordinary listings. This is not a security feature — it is a convention for keeping configuration out of the way.

```
ls          # visible files only
ls -a       # everything, including . and ..
ls -A       # everything except . and ..
```

Your home directory is full of them: `.bashrc`, `.ssh/`, `.gitconfig`, `.profile`.

6.5 Naming files

Rule	Consequence
Case sensitive	<code>Report.txt</code> , <code>report.txt</code> and <code>REPORT.TXT</code> are three different files.
No meaningful extension	<code>.txt</code> is a convention for humans, not a rule the system enforces. Use <code>file</code> to find out what something actually is.
Almost any character allowed	Only <code>/</code> and the null byte are forbidden. Spaces, quotes and newlines are all legal — and all a nuisance.
Leading dot means hidden	See above.
Leading hyphen is trouble	<code>-file</code> looks like an option to most commands. Refer to it as <code>./-file</code> .

Spaces in filenames

A space separates arguments, so `rm my file.txt` tries to delete two files: `my` and `file.txt`. Quote the name (`rm "my file.txt"`) or escape the space (`rm my\ file.txt`). Better still, use underscores or hyphens instead of spaces.

6.6 Building a directory hierarchy

`mkdir` creates one directory; `-p` creates a whole chain of them, making any missing parents along the way.

```
mkdir practice # one directory
mkdir -p practice/dir1/dir2/dir3 # the whole chain in one command
```

Without `-p`, the second command fails: `mkdir` will not create `dir2` inside a `dir1` that does not yet exist.

Several directories at the same level can be given as separate arguments, and brace expansion (Section 11.5.1) makes that compact:

```
mkdir abc lmn klm pqr rst # five siblings
mkdir -p WebApp/{HTML,Assets,Logs} # three subdirectories in one go
mkdir -p WebApp/Assets/{Images,Fonts}
```

6.6.1 A worked hierarchy

Combining the two techniques builds an arbitrary tree in a single command:

```
mkdir -p UoC/{ADMTC,Admin,eLC,CMT,SriPalee,IHRA/Academic}
```

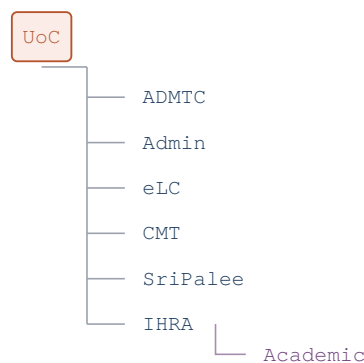


Figure 6.3. The tree produced by the single `mkdir -p` above.

Tip

Verify the result rather than assuming it. `ls -R UoC` lists the tree recursively; `tree UoC` draws it, if the `tree` package is installed.

6.7 Mounting filesystems

A newly created filesystem — a partition, a USB stick, a network share — is not automatically visible. It must be *mounted*: attached to an existing empty directory, called its *mount point*. From then on, that directory *is* the other filesystem.

The terminology dates from tape reels, which had to be physically mounted before their contents could be read. The scheme looks roundabout compared to drive letters, but it has a real advantage: storage can be added, moved or replaced without any path in the system changing.

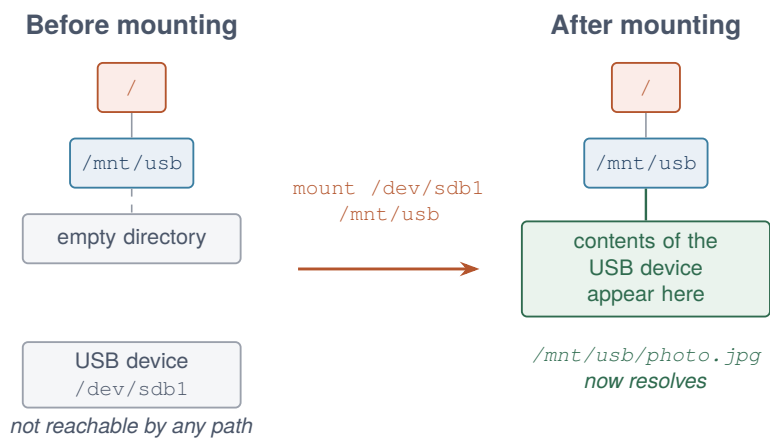


Figure 6.4. Mounting attaches a filesystem to a directory. That directory then acts as the entry point to the device.

Command	Effect
<code>mount</code>	List everything currently mounted.
<code>mount /dev/sdb1 /mnt/usb</code>	Attach that device at that mount point.
<code>mount /mnt/cdrom</code>	Mount an entry already described in <code>/etc/fstab</code> .
<code>umount /mnt/usb</code>	Detach it. Note the spelling: <code>umount</code> , not <code>unmount</code> .
<code>umount /dev/sdb1</code>	The same, named by device instead of mount point.
<code>df -h</code>	Show mounted filesystems with sizes and free space in human-readable units.
<code>lsblk</code>	List block devices and where each is mounted.

Caution

`umount` fails with *device is busy* if any process has a file open on that filesystem — including a shell whose working directory is inside it. `cd` somewhere else first. Never physically remove a device before unmounting it: writes may still be buffered in memory.

Mounting normally needs root

Both `mount` and `umount` require administrative rights, so in practice they are run under `sudo`. Desktop environments mount removable media automatically under `/media/user/`, which is why a USB stick just appears when you plug it in.

6.8 How much space

```
df -h          # free space per mounted filesystem
du -h documents # size of each item under a directory
du -sh documents # a single summary figure for that directory
du -sh * | sort -h # rank everything here by size
```

Going further

The layout in Figure 6.1 is documented in a single manual page: `man hier`. `/etc/fstab` is the file that tells the system what to mount at boot, and reading yours explains why your disks appear where they do. When you are ready for the level below the filesystem, look at partitioning (`lsblk`, `fdisk`) and at what an *inode* is — the structure holding a file’s metadata, which explains why permissions belong to a file and not to its name, and why a hard link is not a copy.

CHAPTER 7

Working with Files and Directories

The core working vocabulary of the shell: listing, creating, copying, moving, deleting and viewing. Roughly a dozen commands, each with a handful of options that matter. This is the material every laboratory exercise in the module is built from.

7.1 Listing — `ls`

Syntax

```
ls [options] [path ...]
```

With no path, lists the current directory. Several paths may be given at once.

Option	Effect
<code>-l</code>	Long listing: type and permissions, link count, owner, group, size, timestamp, name.
<code>-a</code>	All entries, including hidden ones and <code>.</code> and <code>..</code> .
<code>-A</code>	All entries except <code>.</code> and <code>..</code> .
<code>-h</code>	Human-readable sizes (4.0K, 2.3M). Only meaningful with <code>-l</code> .
<code>-R</code>	Recursive — descend into every subdirectory.
<code>-t</code>	Sort by modification time, newest first.
<code>-r</code>	Reverse the sort order.
<code>-S</code>	Sort by file size, largest first.
<code>-d</code>	Show a directory itself rather than its contents.
<code>-i</code>	Show each file's inode number.
<code>-1</code>	One entry per line.

```
ls                # plain listing of the current directory
ls -l             # long listing
ls -lh            # long listing, human-readable sizes
ls -la            # long listing including hidden files
ls -lah /etc      # all three, applied to /etc
ls /bin /sbin     # two directories in one command
ls ~              # your home directory, from anywhere
ls -l ..          # the parent directory, without moving there
ls -lR ~/practice # a whole tree
```

7.1.1 Reading a long listing

```
$ ls -l report.txt
-rw-r--r-- 1 student staff 2048 Aug  4 10:15 report.txt
```

Field	Meaning
-rw-r--r--	File type, then the three permission triplets. See Chapter 8.
1	Number of hard links to the file. For a directory, the number of entries it contains, plus two.
student	Owning user.
staff	Owning group.
2048	Size in bytes.
Aug 4 10:15	Last modification time.
report.txt	Name.

7.2 Creating

7.2.1 Directories — `mkdir`

Covered in Section 6; the essentials again for reference:

```
mkdir reports           # one directory
mkdir dir1 dir2 dir3    # three at the same level
mkdir -p a/b/c          # a chain, creating parents as needed
mkdir -p WebApp/{HTML,Assets,Logs} # several branches at once
mkdir -m 755 shared     # create with explicit permissions
```

7.2.2 Empty files — `touch`

`touch` creates a file if it does not exist, and updates its timestamps if it does.

```
touch alpha.txt          # create one empty file
touch alpha.txt beta.txt # create two, in one command
touch report.txt log.txt data.csv config.ini # or four
touch home.html contact.html # the efficient way: never one at a time

touch -d "2 days ago" beta.txt # set the timestamp to two days back
touch -t 202608041015 report.txt # set it to a literal YYYYMMDDhhmm
touch -a file.txt            # update only the access time
touch -m file.txt            # update only the modification time
```

Tip

Whenever a task says "create these files", give them all to `touch` as arguments in a single command. The same applies to `mkdir`: one command, many arguments, is always the intended answer.

7.2.3 Files with content, without an editor

Three ways, each with a different use.

```
# 1. echo with redirection -- one line, quickest
echo "Hello Linux" > data.txt
echo "<h2>Home Page</h2>" > home.html
echo "<p>Contact us at support@example.com</p>" > contact.html

# 2. append a second line without erasing the first
echo "Permissions are important" >> report.txt

# 3. cat with redirection -- several lines typed interactively
cat > logfile.txt
Pgsql
Copy
Edit
Login
Logout
Restart
Shutdown
Sleep
      (press Ctrl+D on a line of its own to finish)
```

Caution

> truncates: it empties an existing file before writing. >> appends. Getting these the wrong way round destroys the file's previous contents with no warning and no undo.

7.2.4 Building a multi-line file in one command

Three equivalent ways to write the three lines of a report:

```
# separate appends
echo "Linux File Handling" > report.txt
echo "Permissions are important" >> report.txt
echo "Access Control is Key" >> report.txt

# one echo with escape interpretation
echo -e "Linux File Handling\nPermissions are important\nAccess Control is Key" \
  > report.txt

# a here-document
cat > report.txt <<EOF
Linux File Handling
Permissions are important
Access Control is Key
EOF
```

7.3 Copying — cp

Syntax

```
cp [options] <source> <destination>
cp [options] <source ...> <directory>
```

Option	Effect
<code>-r</code>	Recursive — required to copy a directory and its contents.
<code>-i</code>	Interactive: ask before overwriting an existing file.
<code>-n</code>	Never overwrite.
<code>-v</code>	Verbose: report each file as it is copied.
<code>-p</code>	Preserve timestamps, ownership and permissions.
<code>-u</code>	Copy only when the source is newer than the destination.

```
cp sample.txt samplecopy.txt      # copy to a new name, same directory
cp beta.txt backup_beta.txt       # the same idea
cp report.txt ~/backups/          # copy into a directory, keeping the name
cp -r folder1 folder2             # copy a whole directory tree
cp *.log backup/                  # every .log file into backup/
cp main.ttf Logs/backup_main.ttf  # copy and rename in one step
```

Note

If the destination is an existing *directory*, the file is copied *into* it under its own name. If the destination does not exist, it is treated as the new file's name. That single rule explains most surprising `cp` results.

7.4 Moving and renaming — `mv`

There is no separate rename command: moving a file to a new name within the same directory *is* renaming it.

```
mv demo newdemo                  # rename
mv alpha.txt alpha_data.txt      # rename
mv report.txt ~/archive/         # move into a directory
mv *.txt archive_records/        # move everything matching a pattern
mv config.ini ~/conf.bak         # move and rename in one command
mv report_old.txt archives/report_final.txt # likewise
mv -i old.txt new.txt            # ask before overwriting
mv -v *.log logs/                # report what is being moved
```

Caution

`mv` onto an existing file replaces it permanently, with no prompt by default. The previous file cannot be recovered. Use `-i` when the destination might already exist.

7.5 Deleting

7.5.1 Files — `rm`

```
rm notes.txt                    # delete one file
rm *.tmp                        # delete every file matching the pattern
rm -i important.txt             # ask for confirmation first
rm -f missing.txt              # force: no prompt, no error if absent
rm -v *.log                     # report each deletion
```

7.5.2 Empty directories — `rmdir`

```
rmdir trash          # only works if trash is empty
rmdir -p a/b/c       # remove c, then b, then a, while each becomes empty
```

7.5.3 Non-empty directories — `rm -r`

`rmdir` refuses to remove a directory with anything in it:

```
$ rmdir newfolder1
rmdir: failed to remove 'newfolder1': Directory not empty
```

The correct command in that case is `rm` with `-r`:

```
rm -r practice      # remove the directory and everything inside it
rm -ri practice     # the same, confirming each item
```

There is no recycle bin

`rm` is permanent. The shell has no undo and no trash folder — files deleted from the command line are gone.

`rm -rf` is the most destructive command on the system, because `-f` suppresses exactly the questions that would have saved you. Before running any `rm` with a wildcard, run the same pattern through `ls` first and read what it matches:

```
ls *.tmp           # check what the pattern matches
rm *.tmp           # only then delete
```

7.5.4 Deleting through a tree

To remove every `.tmp` file in a directory and all its subdirectories, `rm *.tmp` is not enough — a plain wildcard does not descend. Use `find`:

```
find testdir -name "*.tmp" -type f -delete

# safer: look first, delete second
find testdir -name "*.tmp" -type f
find testdir -name "*.tmp" -type f -exec rm {} +
```

7.6 Viewing file contents

Command	Shows
<code>cat file.txt</code>	The whole file at once. Fine for short files.
<code>cat a.txt b.txt</code>	Both files, concatenated — which is what the name means.
<code>tac file.txt</code>	The whole file with the lines in reverse order.
<code>head file.txt</code>	The first 10 lines.
<code>head -n 5 file.txt</code>	The first 5 lines.
<code>tail file.txt</code>	The last 10 lines.
<code>tail -n 20 file.txt</code>	The last 20 lines.
<code>tail -f server.log</code>	The last lines, then new ones as they are written. The standard way to watch a log.
<code>less file.txt</code>	A pager: scroll with the arrow keys, search with <code>/pattern</code> , quit with <code>q</code> . Use this for anything long.
<code>more file.txt</code>	An older, simpler pager.
<code>wc -l file.txt</code>	Counts lines. <code>-w</code> words, <code>-c</code> bytes.

```
cat data.txt                # display a file
cat file1.txt file2.txt     # display two, one after the other
cat file1.txt file2.txt > combined.txt # and save the result
cat info1.html info2.html info3.html > all_info.html
tac logfile.txt             # reverse the line order on screen
```

7.7 Identifying files — `file`

The extension is only a convention. `file` inspects the contents and reports what something actually is.

```
$ file /bin/cat
/bin/cat: ELF 64-bit LSB pie executable, x86-64, dynamically linked

$ file /etc/passwd
/etc/passwd: ASCII text

$ file /usr/bin/passwd
/usr/bin/passwd: setuid ELF 64-bit LSB pie executable, x86-64
```

The three above are worth comparing: two executables and one plain text file, where the third is marked `setuid` — a detail explained in Section 8.6.

7.8 Wildcards

A *wildcard* (or *glob*) is a pattern the shell expands into a list of matching filenames *before* the command runs. The command itself never sees the pattern.

Pattern	Matches
<code>*</code>	Any number of characters, including none. On its own, every file in the directory.
<code>?</code>	Exactly one character.
<code>A*</code>	Any name beginning with <code>A</code> .
<code>A*txt</code>	Any name beginning with <code>A</code> and ending with <code>txt</code> .
<code>List??</code>	<code>List</code> followed by exactly two characters.
<code>A??f</code>	<code>A</code> , then exactly two characters, then <code>f</code> .
<code>[abc]*</code>	Any name starting with <code>a</code> , <code>b</code> or <code>c</code> .
<code>[a-d]?</code>	A name starting with <code>a</code> to <code>d</code> , followed by exactly one character.
<code>[[:upper:]]*</code>	Any name beginning with an uppercase letter.
<code>[[:digit:]]*</code>	Any name beginning with a digit.
<code>[![:digit:]]*</code>	Any name <i>not</i> beginning with a digit.
<code>[!abc]*</code>	Any name not beginning with <code>a</code> , <code>b</code> or <code>c</code> .

7.8.1 Character classes

Inside square brackets, the POSIX classes are more readable than ranges and behave correctly regardless of locale:

<code>[[:alpha:]]</code>	letters	<code>[[:digit:]]</code>	digits
<code>[[:alnum:]]</code>	letters and digits	<code>[[:space:]]</code>	whitespace
<code>[[:upper:]]</code>	uppercase letters	<code>[[:lower:]]</code>	lowercase letters
<code>[[:punct:]]</code>	punctuation	<code>[!...]</code>	negates the set

7.8.2 Wildcards in practice

```
ls *           # everything here
ls L*         # names starting with L
ls *lab*      # names containing "lab" anywhere
ls [LM]*      # names starting with L or M
ls [[:upper:]]* # names starting with a capital letter
ls temp*.csv  # starts with temp, ends with .csv
ls ????.sh   # exactly four characters, then .sh
rm draft[0-9].html # draft1.html, draft2.html -- but not draft_notes.html
cp *.log backup/ # every log file into backup/
chmod 640 *.txt  # one command, every text file
```

Case sensitivity applies here too

`ls *lab*` will not find `MSCLabs.txt`, because the capital `L` does not match the lowercase `l` in the pattern. Use `ls *[Ll]ab*` to match either, or `shopt -s nocaseglob` to make globbing case-insensitive for the current shell.

When a pattern matches nothing

Bash leaves an unmatched pattern untouched and passes it through literally, so `ls *.xyz` reports `No such file or directory: *.xyz` rather than staying silent. In a script this can lead a loop to run once with the pattern itself as its value — `shopt -s nullglob` makes an unmatched pattern expand to nothing instead.

7.9 Navigating between directories

Beyond `cd`, the shell keeps a stack of directories, which is useful when you need to go somewhere, do something and come straight back.

Command	Effect
<code>pushd /usr</code>	Push the current directory onto the stack and change to <code>/usr</code> . Prints the stack.
<code>pushd /var/log</code> <code>popd</code>	Push again and move to <code>/var/log</code> . The stack now holds three entries. Remove the top entry and change to whatever is now on top — that is, go back one step.
<code>dirs -v</code>	Show the stack, numbered.
<code>cd -</code>	Simpler alternative: toggle between the current and previous directory.

```
$ pwd
/home/student
$ pushd /usr
/usr /home/student
$ pushd /var/log
/var/log /usr /home/student
$ popd
/usr /home/student
$ popd
/home/student
```

Tip

`pushd` and `popd` are Bash builtins, so they are documented in `man bash` rather than in a page of their own. Search within it by typing `/pushd` once the pager opens.

7.10 Combining commands on one line

Three operators join commands, and the difference between them is whether the next command depends on the previous one succeeding.

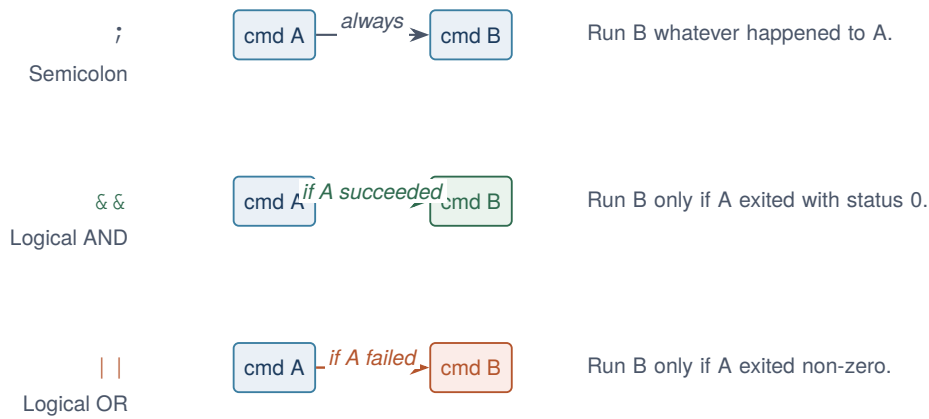


Figure 7.1. Every command returns an *exit status*: zero for success, non-zero for failure. `&&` and `||` branch on it.

```
cd /tmp ; ls ; pwd                # three commands, run in sequence

mkdir labtest && echo "Created successfully"    # message only on success
mkdir labtest || echo "Failed to create"       # message only on failure

# both branches in a single line
mkdir labtest && echo "Created successfully" || echo "Failed to create"

cd /nonexistent && rm -rf *           # the guard that prevents a disaster
```

Caution

That last line is the reason `&&` matters. Written with `;` instead, the `cd` fails, the `rm` runs anyway — and deletes everything in whichever directory you happened to be standing in.

7.11 Counting and combining output

```
ls -R WebApp                      # every file, recursively, with headers
find WebApp -type f               # just the paths, one per line
find WebApp -type f | wc -l       # how many files in total

ls -l > permissions_check.txt     # save a listing to a file
ls -l >> permissions_check.txt     # append instead of overwriting
```

Redirection and pipes are covered properly in Chapter 11; the two forms above are enough for the exercises in this chapter.

7.12 A worked structure

Bringing the chapter together — building, populating and reorganising a small project tree.

```
# structure, in as few commands as possible
mkdir -p WebApp/{HTML,Assets/{Images,Fonts},Logs}
touch WebApp/HTML/{home.html,contact.html}
touch WebApp/Assets/Images/logo.png WebApp/Assets/Fonts/main.ttf

# content, without an editor
```

```

echo "This is a web application project with images, fonts, and HTML pages." \
    > WebApp/readme.txt
echo "<h2>Home Page</h2>" > WebApp/HTML/home.html
echo "<p>Contact us at support@example.com</p>" > WebApp/HTML/contact.html

# copy and rename in one step; move to a new directory
cp WebApp/Assets/Fonts/main.ttf WebApp/Logs/backup_main.ttf
mkdir -p WebApp/Media
mv WebApp/Assets/Images/logo.png WebApp/Media/

# selective deletion with a wildcard
cd WebApp/HTML
touch draft1.html draft2.html draft_notes.html draft_backup.html
rm draft[0-9].html          # removes only draft1 and draft2

# inspect and count
ls -laR ~/WebApp
find ~/WebApp -type f | wc -l

# concatenate several files into one
touch info1.html info2.html info3.html
cat info1.html info2.html info3.html > all_info.html

```

Going further

`find` is the command that repays study most: it can select files by name, size, age, type, owner or permissions, and act on the results with `-exec`. Pair it with `xargs` for efficiency on large sets. After that, `grep` (search inside files), `sed` (stream editing) and `awk` (field-oriented processing) form the classic text-processing trio — Chapter 11 introduces them. Also worth knowing: `ln -s` creates symbolic links, and `rsync` copies trees far more capably than `cp -r`.

CHAPTER 8

Accounts, Groups and Permissions

Linux is multi-user by design, and every file carries an owner, a group and a set of permissions. This chapter covers the account system, then the permission model and the two notations — symbolic and numeric — for changing it.

8.1 Users and groups

Every process runs as some user, and every file belongs to one user and one group. Three categories matter:

Account type	Notes
root (UID 0)	The superuser. Bypasses every permission check. Used only for administration, never for ordinary work.
Regular users	Human accounts, each with a home directory and a login shell. UIDs usually start at 1000.
System accounts	Non-human accounts owned by services — <code>www-data</code> , <code>sshd</code> , <code>daemon</code> . They usually have no valid login shell, which limits the damage if the service is compromised.

A *group* is a named set of users. Rather than granting a permission to twenty people individually, you create a group, add the twenty people to it and grant the permission once. Every user has one *primary group* and any number of *supplementary groups*.

8.1.1 Where accounts are stored

File	Contains
<code>/etc/passwd</code>	One line per account: username, UID, primary GID, home directory, login shell. World-readable — which is why the passwords are not in it, despite the name.
<code>/etc/shadow</code>	The hashed passwords and ageing information. Readable only by root.
<code>/etc/group</code>	Group names, GIDs and supplementary members.

```
$ head -1 /etc/passwd
root:x:0:0:root:/root:/bin/bash

$ grep ^student /etc/passwd
student:x:1000:1000:Student,,,:/home/student:/bin/bash
```

The `x` in the second field means "the password hash is in `/etc/shadow`".

8.1.2 Finding out who you are

Command	Reports
<code>whoami</code>	The current username.
<code>id</code>	UID, primary GID and every group you belong to.
<code>groups</code>	Just the group names.
<code>id student</code>	The same, for another user.
<code>who</code>	Everyone currently logged in.
<code>w</code>	Who is logged in and what they are running.

8.1.3 Managing accounts

All of these require root, so in practice they are run under `sudo`.

Command	Effect
<code>useradd bob</code>	Create the account <code>bob</code> , minimally.
<code>useradd -m -s /bin/bash bob</code>	Create it <i>with</i> a home directory (<code>-m</code>) and a login shell (<code>-s</code>).
<code>adduser bob</code>	Debian/Ubuntu wrapper that does all of the above interactively and prompts for a password.
<code>passwd bob</code>	Set or change <code>bob</code> 's password.
<code>passwd</code>	Change your own password.
<code>usermod -aG staff bob</code>	Add <code>bob</code> to the supplementary group <code>staff</code> .
<code>usermod -g staff bob</code>	Change <code>bob</code> 's <i>primary</i> group.
<code>usermod -L bob</code>	Lock the account. <code>-U</code> unlocks it.
<code>userdel bob</code>	Delete the account. <code>-r</code> also removes the home directory.

Caution

In `usermod -aG`, the `-a` means *append*. Leave it out and the user's supplementary groups are *replaced* by the list you gave — which silently removes them from every other group, including `sudo`.

8.1.4 Managing groups

`groupadd` creates a group. Groups exist so that permissions can be granted to a set of users at once instead of to each user separately, which is what makes permission management tractable on a system with many accounts.

Command	Effect
<code>groupadd developers</code>	Create a group.
<code>groupadd -g 1500 developers</code>	Create it with a specific GID.
<code>groupdel developers</code>	Delete it.
<code>gpasswd -a bob developers</code>	Add <code>bob</code> to it.
<code>gpasswd -d bob developers</code>	Remove <code>bob</code> from it.
<code>getent group developers</code>	Show the group and its members.

Note

Group membership is established at login. After being added to a group you must log out and back in — or run `newgrp developers` — before the new membership takes effect in your current session.

8.1.5 Becoming another user

Command	Effect
<code>su bob</code>	Switch to <code>bob</code> , keeping the current environment. Requires <code>bob</code> 's password.
<code>su - bob</code>	Switch to <code>bob</code> with a full login shell — their environment, their home directory.
<code>su -</code>	Become root, if root has a password set.
<code>sudo command</code>	Run one command as root, authenticating with <i>your own</i> password. Requires membership of <code>sudo</code> or <code>wheel</code> .
<code>sudo -i</code>	Start an interactive root shell.
<code>exit</code>	Return to your own account.

Tip

`sudo` is preferable to `su` for administration: it needs no shared root password, it grants privilege one command at a time, and it logs every use to `/var/log/auth.log`. On Ubuntu the root account has no password at all by design, so `su -` simply will not work.

8.2 The permission model

Every file and directory carries nine permission bits: three permissions (*read*, *write*, *execute*) for each of three classes of user (*user*, *group*, *other*).

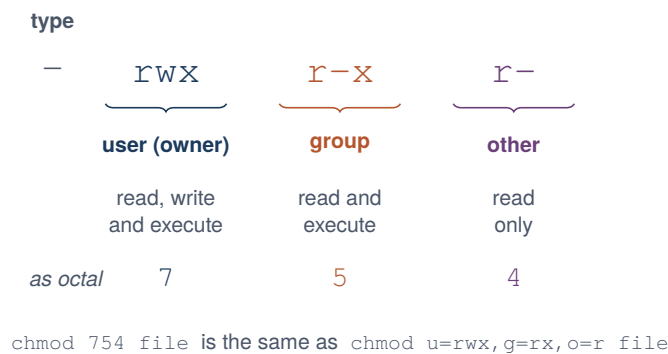


Figure 8.1. The ten characters of a long listing: one type character followed by three triplets.

8.2.1 The three classes

Letter	Class	Who
u	User	The owner of the file — normally whoever created it.
g	Group	Members of the file’s owning group.
o	Other	Everyone else on the system.
a	All	Shorthand for u, g and o together.

The classes are checked in order and the *first* match wins. If you own a file and the user bits say no, you are denied — even if the group bits would have allowed it.

8.2.2 The three permissions — and what they mean for directories

This is the part that catches people out. The same three letters mean different things depending on whether the target is a file or a directory.

Perm	On a file	On a directory
r (4)	Read the contents.	List the names it contains (<code>ls</code>).
w (2)	Modify the contents.	Create, rename and delete entries inside it.
x (1)	Run it as a program.	Enter it (<code>cd</code>) and access things through it.
–	Permission not granted.	Permission not granted.

Three consequences worth remembering

- A directory with `r` but no `x` lets you list the names but not read any of the files. Nearly useless.
- A directory with `x` but no `r` lets you open a file whose exact name you already know, but not discover what is there. This is how a home directory can be made traversable without being browsable.
- Deleting a file depends on the *directory’s* write permission, not the file’s. You can delete a read-only file from a directory you can write to — and you cannot delete a file you own from a directory you cannot write to.

8.3 Changing permissions — `chmod`

Two notations, both producing the same result.

8.3.1 Symbolic notation

Syntax

`chmod [who][operator][permissions] <file>`

who is any of `u g o a`; *operator* is `+` (add), `-` (remove) or `=` (set exactly); *permissions* is any of `r w x`.

Op	Effect
+	Add the listed permissions, leaving the others untouched.
-	Remove the listed permissions, leaving the others untouched.
=	Set the permissions to exactly this list, clearing anything not named.

```
chmod u+x script.sh      # give the owner execute permission
chmod g-w report.txt     # take write away from the group
chmod o+r notes.txt      # let everyone read it
chmod a+r public.txt     # all three classes can read
chmod u=rwx,g=rx,o=r file # set all nine bits explicitly
chmod g-x,o+r /root/xyz  # two changes in one command
chmod u=rx,o=rx Academic # user and other: read and execute only
chmod u=rw,g=,o= project.txt # owner read/write, nothing for anyone else
chmod -R g+w shared/     # apply through a whole directory tree
```

Tip

Symbolic notation is the safer choice when you want to change one thing. `chmod u+x script.sh` adds execute for the owner and touches nothing else; the numeric equivalent would require you to know and restate all nine bits.

8.3.2 Numeric (octal) notation

Each permission has a value; add them up per class.

Digit	Bits	Permissions	Sum of
0	---	No privileges	—
1	--x	Execute	1
2	-w-	Write	2
3	-wx	Write and execute	2 + 1
4	r--	Read	4
5	r-x	Read and execute	4 + 1
6	rw-	Read and write	4 + 2
7	rwx	Read, write and execute	4 + 2 + 1

Three digits, in the order user, group, other.

```
chmod 755 script.sh      # rwx for owner, r-x for group and other
chmod 644 report.txt     # rw- for owner, r-- for group and other
chmod 700 private/       # rwx for owner, nothing for anyone else
```

```

chmod 640 report.txt      # rw- owner, r-- group, --- other
chmod 600 ~/.ssh/id_rsa   # rw- owner only -- required for SSH private keys
chmod 664 archive.txt     # rw- owner, rw- group, r-- other
chmod 400 project.txt     # read-only for the owner, nothing for anyone else
chmod 444 readonly.txt    # read-only for absolutely everyone
chmod 4XX file            # user read-only, in the pattern used by exercises

```

8.3.3 Common patterns

Mode	String	Typical use
644	rw-r--r--	Ordinary documents: owner edits, everyone reads.
755	rxwxr-xr-x	Programs, scripts and most directories.
600	rw-----	Private files: SSH keys, credentials.
700	rxw-----	Private directories, including <code>~/.ssh</code> .
664	rw-rw-r--	Files a team edits together.
775	rxwxrwxr-x	Directories a team writes to.
444	r--r--r--	Read-only for everyone, owner included.
777	rxwxrwxrwx	Everything for everyone. Almost always wrong.

Caution

`chmod 777` is not a fix, it is a way of not diagnosing the problem. It lets any user on the system modify or replace the file. If something is not working, find out which class is being denied — `ls -l` and `id` will tell you — and grant only that.

8.3.4 Reading a mode back

```

ls -l report.txt          # the symbolic form
stat -c "%a %n" report.txt # the numeric form
stat report.txt           # everything, both notations
ls -l > permissions_check.txt # save a listing for the record

```

```

$ ls -l
-rw-r--r-- 1 student staff 120 Aug  4 10:15 report.txt
-rwxr-xr-- 1 student staff 2048 Aug  4 10:16 run.sh
$ stat -c "%a %n" report.txt run.sh
644 report.txt
754 run.sh

```

Reading these back: `-rw-r--r--` is a regular file the owner can read and write and everyone else can only read. `-rwxr-xr--` is a regular file the owner can read, write and execute, the group can read and execute, and others can only read.

8.4 Changing ownership

Command	Effect
<code>chown bob file.txt</code>	Change the owning user. Root only.
<code>chown bob:staff file.txt</code>	Change owner and group together.
<code>chown :staff file.txt</code>	Change only the group.
<code>chgrp staff file.txt</code>	The same, using the dedicated command.
<code>chown -R bob:staff project/</code>	Apply through a directory tree.

Note

Only root can give a file away. An ordinary user may change a file's *group*, but only to a group they themselves belong to — otherwise handing over ownership would be a way of evading disk quotas.

8.5 Default permissions — `umask`

New files do not appear with arbitrary permissions. The system starts from a base — 666 for files, 777 for directories — and removes whatever bits the *umask* specifies.

umask	New file	New directory	Effect
022	644	755	The usual default: others can read.
002	664	775	Group can write too — for shared work.
077	600	700	Private: nobody else sees anything.

```
umask          # show the current value
umask 077      # set it for this shell session only
```

To make a change permanent, add the `umask` line to `~/.bashrc`.

Note

Files never get the execute bit by default — which is why a newly written script must be given `chmod +x` before it will run.

8.6 The special bits

Beyond the nine, three further bits exist. They appear as a fourth leading octal digit.

Bit	Octal	Effect
setuid	4000	An executable runs with the privileges of its <i>owner</i> rather than the person running it. This is how <code>/usr/bin/passwd</code> lets an ordinary user write to <code>/etc/shadow</code> . Shows as <code>s</code> in the user triplet.
setgid	2000	On an executable, runs with the group's privileges. On a <i>directory</i> , far more useful: new files inside inherit the directory's group, which is how a shared project directory stays shared.
sticky	1000	On a directory, only the owner of a file may delete it, regardless of the directory's write permission. <code>/tmp</code> uses this — it is world-writable, yet you cannot delete another user's files. Shows as <code>t</code> .

```
$ ls -ld /tmp /usr/bin/passwd
drwxrwxrwt 12 root root 4096 Aug  4 09:02 /tmp
-rwsr-xr-x  1 root root 68208 Mar 23 14:11 /usr/bin/passwd
```

```
chmod 2775 shared/      # setgid: new files inherit the group
chmod g+s shared/      # the same, symbolically
chmod 1777 /tmp         # sticky bit
chmod +t shared/        # the same, symbolically
```

Caution

Never set the setuid bit on a script you have written. It is a well-known route to privilege escalation, and most modern kernels ignore it on scripts for exactly that reason.

8.7 Worked permission changes

The exercises phrase permission tasks in prose. Translating that prose is the skill.

Requirement	Command
Read-only for all users	<code>chmod 444 project.txt</code>
Owner read + write, group read, others nothing	<code>chmod 640 *.txt</code>
Owner all, group and others write only	<code>chmod 722 Arts.txt</code>
Owner read/write, group read/write, others read	<code>chmod 664 *.txt</code>
Only the owner may read, write and execute	<code>chmod 700 script.sh</code>
User read only, using numbers	<code>chmod 400 ADMTC</code>
Other write only, using numbers	<code>chmod 002 eLC</code>
User and other read + execute, using numbers	<code>chmod 505 Admin</code>
User and other read + execute, using letters	<code>chmod u=rx,o=rx Academic</code>
Read and write, using letters	<code>chmod u=rw,g=rw,o=rw CMT</code>

Requirement	Command
Owner read/write only, nothing for group or others	<code>chmod 600 main.ttf</code>
Owner read/write but not execute; nothing for others	<code>chmod 600 Logs</code>
Add execute for owner, remove everything for group and others	<code>chmod u+x,g=,o= science.txt</code>
Remove owner read, add group and other execute	<code>chmod u-r,g+x,o+x ClassA.txt</code>

Numeric mode always sets all three classes

This is the trap in the rows above that say “using numbers”. A requirement such as “*change the permissions of user of ADMTC to read only*” names only one class — but `chmod` with a numeric argument rewrites all nine bits at once. `chmod 400 ADMTC` therefore also strips group and other to nothing, which the requirement never asked for.

To change one class and genuinely leave the rest alone, you must either read the current mode first and reconstruct all three digits from it:

```
stat -c "%a %n" ADMTC      # e.g. 755  -> user must become 4, so use 455
chmod 455 ADMTC
```

or use the symbolic form, which touches only the classes you name:

```
chmod u=r ADMTC           # group and other keep whatever they had
```

The numeric answers in the table assume the unnamed classes should end up with no permissions. Say which reading you have used when you write the exercise up.

Note

For the same reason, `chmod u=rx,o=rx Academic` is the faithful translation of “user and other read & execute”: it changes user and other and leaves the group exactly as it was. No three-digit numeric command can express that.

Going further

Standard Unix permissions can only express one owner and one group. When you need finer control — “these three specific users may write, nobody else” — the answer is *access control lists*: `getfacl` and `setfacl`. A file with an ACL shows a + at the end of its mode string in `ls -l`. Beyond that lie the mandatory access control systems, SELinux and AppArmor, which constrain what a *program* may do regardless of who runs it.

CHAPTER 9

The Vim Editor

Vim is on every Unix-like system, works over the slowest remote connection and needs no graphical environment. It is also modal and composable, which is why it feels wrong for the first hour and fast forever afterwards. This chapter covers enough to be productive, then the features that make Vim worth choosing over a plain editor.

9.1 Why Vim

You will meet Vim whether or not you choose to. It is the editor guaranteed to be installed on a minimal server, the one that opens when a tool needs you to write a commit message or edit a configuration file, and the one that still works when a graphical editor cannot — over SSH, in single-user mode, on a system that will not boot properly.

Beyond availability, two design choices make it worth learning deliberately:

It is modal

Moving and changing text is separated from typing text, so every key on the keyboard is free to be a command rather than a character.

It composes

Commands are built from small parts that combine, so learning one new motion immediately makes every operator more useful. This is the idea developed in Section 9.6, and it is what people mean when they say Vim is a language rather than a set of shortcuts.

9.1.1 Vim and vi

`vi` is the original 1976 editor and the one named in the POSIX standard. Vim — *Vi IMproved* — is a rewrite that keeps every `vi` command working and adds a great deal on top.

Vim adds	Covered in
Multi-level undo and redo	Section 9.8
Visual mode	Section 9.9
Text objects	Section 9.6.1
Named registers and clipboard access	Section 9.10
Recordable macros	Section 9.12
Split windows, buffers and tabs	Section 9.13
Syntax highlighting, and a configuration file	Section 9.14
Plugins	Section 9.15

On almost every current distribution, typing `vi` starts Vim: the name is a symbolic link, or a small `vi`-compatible build. Check which you have:

```
$ vim --version | head -1
VIM - Vi IMproved 9.0 (2022 Jun 28, compiled Aug 01 2024)

$ readlink -f "$(which vi)"
/usr/bin/vim.basic
```

When a laboratory task says “vi”

Treat it as “the Vim on this machine”. Everything in the exercises — opening a file, inserting, saving, quitting — is identical either way, and anything you type as `vi` runs Vim. The one thing to watch for is *compatible mode*: if Vim is started as `vi` it may disable the improvements for strict compatibility, which is why multi-level undo and the arrow keys sometimes misbehave. `:set nocompatible`, or simply having a `~/vimrc`, turns them back on.

Tip

If Vim is not installed — some minimal container images ship only `vi` from BusyBox — install it with `sudo apt install vim` (Chapter 10).

9.2 Modes — the central idea

Most editors have one mode: keystrokes insert characters. Vim separates *moving and changing* text from *typing* it, and gives each its own mode. That is why pressing `h` sometimes writes an `h` and sometimes moves the cursor left.

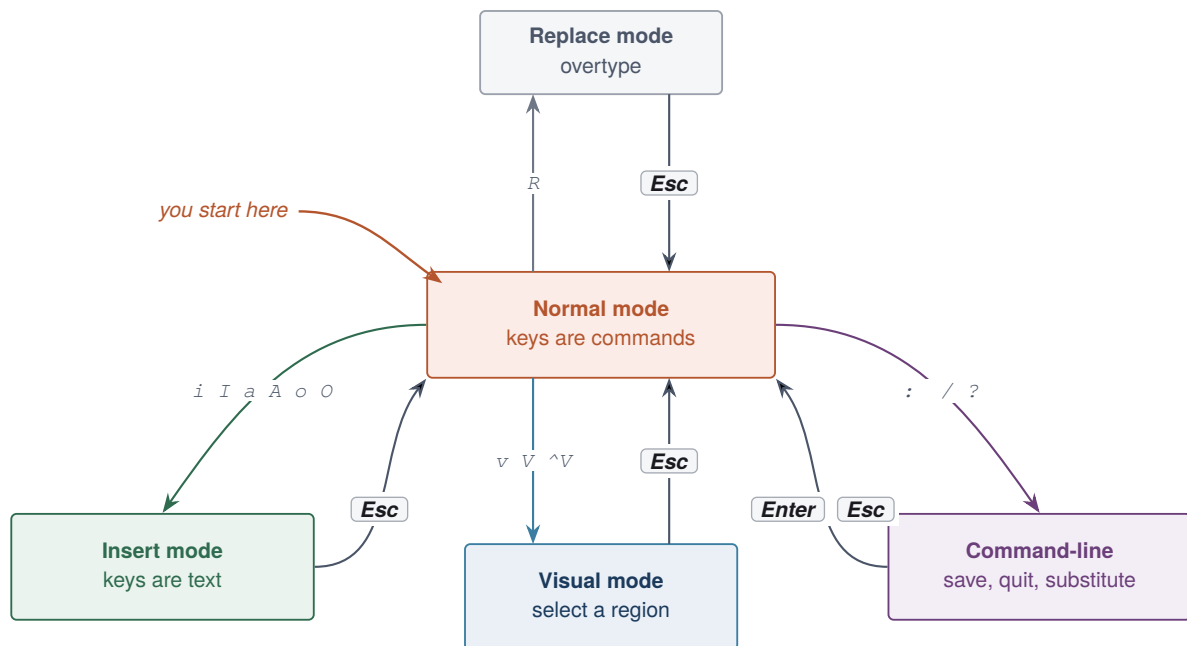


Figure 9.1. `Esc` always returns to Normal mode. When you are lost, press it twice and start again.

Mode	What keys do
Normal	Run commands: move, delete, copy, paste. Where you spend most of your time, and where Vim always starts.
Insert	Type text, as in any other editor.
Visual	Select a region, then apply one operator to all of it.
Command-line	Everything after <code>:</code> , <code>/</code> or <code>?</code> — saving, quitting, substitution, searching.
Replace	Overtyping rather than insert. Entered with <code>R</code> ; rarely needed.

Tip

The rule that resolves all confusion: `Esc` returns to Normal mode from anywhere. If a keystroke does something unexpected, you were in the wrong mode. Press `Esc`, then `u` to undo, and try again. Vim shows the current mode at the bottom left — `-- INSERT --`, `-- VISUAL --` — and shows nothing at all in Normal mode.

9.3 Opening, saving and quitting

Command	Effect
<code>vim filename</code>	Open the file, or create it if it does not exist.
<code>vim +25 filename</code>	Open it with the cursor on line 25.
<code>vim +/word file</code>	Open it at the first occurrence of <code>word</code> .
<code>vim file1 file2</code>	Open several files; <code>:n</code> moves to the next.
<code>vim -R filename</code>	Open read-only. <code>view filename</code> does the same.
<code>vimdiff a.txt b.txt</code>	Open two files side by side with the differences highlighted.

These are typed in Normal mode and begin with a colon, which puts you into command-line mode:

Command	Effect
<code>:w</code>	Write (save) and carry on editing.
<code>:w newname</code>	Save under a different name.
<code>:wq</code> or <code>:x</code>	Write and quit. <code>:x</code> writes only if something changed.
<code>ZZ</code>	Write and quit, without the colon.
<code>:q</code>	Quit — refused if there are unsaved changes.
<code>:q!</code>	Quit, discarding all changes.
<code>ZQ</code>	The same, without the colon.
<code>:wqa</code>	Write and quit every open file.
<code>:e!</code>	Reload the file from disk, discarding changes.
<code>:e otherfile</code>	Open another file in this window.

Caution

`:q!` is unrecoverable. If you are not sure whether you want the changes, `:w backup.txt` first and then decide.

Opening a root-owned file you forgot to `sudo`

You have edited `/etc/ssh/sshd_config` as an ordinary user and `:w` reports a read-only file. Rather than losing the edit, write it through `sudo` from inside Vim:

```
:w !sudo tee % > /dev/null
```

`%` stands for the current file name. Vim will then notice the file has changed on disk and offer to reload it — answer `L`.

9.4 Moving around

All of these are Normal-mode keys. Learning the `h j k l` cluster is worth the initial awkwardness — your hand never leaves the home row.

Key	Moves	Key	Moves
h	Left one character	0	Start of line
j	Down one line	^	First non-blank character
k	Up one line	\$	End of line
l	Right one character	gg	First line of the file
w	Forward one word	G	Last line of the file
b	Back to the start of a word	25G	Line 25
e	Forward to the end of a word	:25	Line 25
W B E	The same, but words are whitespace-separated	H M L	Top, middle, bottom of the screen
{	Previous paragraph	Ctrl+F	Forward one screen
}	Next paragraph	Ctrl+B	Back one screen
%	Jump to the matching bracket	Ctrl+D	Half a screen down, up
		Ctrl+U	
f x	Forward to the next <i>x</i> on this line	t x	Forward to just before it
F x	Backward to the previous <i>x</i>	; ,	Repeat that jump forward, backward
{ }		"	Back to where you just jumped from

Ctrl+G shows the file name, the current line and the total. :num displays the current line number.

Counts

Almost every command accepts a leading number, meaning “do this that many times”. 5j moves down five lines, 3dd deletes three lines, 10x deletes ten characters. This is what makes Vim fast, and it works with everything in the next section.

9.4.1 Marks

A *mark* is a saved position you can jump back to.

ma	Set mark <i>a</i> at the cursor. Any letter will do.
'a	Jump to the start of the line holding mark <i>a</i> .
`a	Jump to the exact position of mark <i>a</i> .
"	Jump back to where you were before the last big jump.
:marks	List every mark currently set.

9.5 Entering insert mode

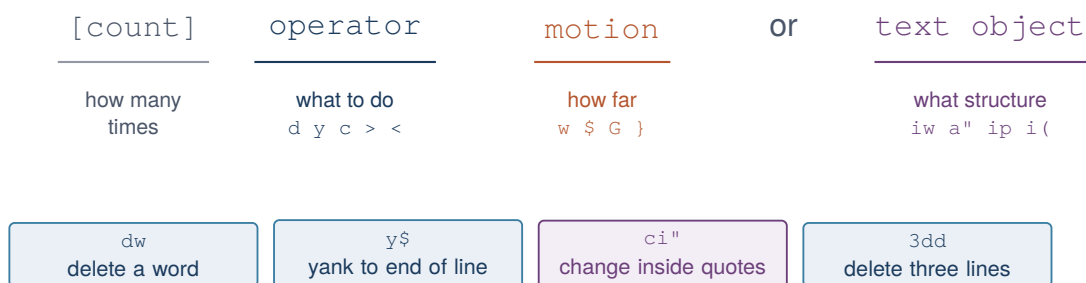
Six ways in, each putting the cursor somewhere different. Choosing the right one saves a separate movement.

Key	Starts inserting
i	Before the cursor.
I	At the beginning of the current line (first non-blank).
a	After the cursor (<i>append</i>).
A	At the end of the current line.
o	On a new line <i>below</i> the current one.
O	On a new line <i>above</i> the current one.

Esc returns to Normal mode.

9.6 The grammar: operator + motion

This is the idea that makes Vim different from a set of shortcuts. A command is built from an *operator* and a *motion*, optionally preceded by a count:



learn one new motion and every operator gains a new use

Figure 9.2. Operators and motions multiply. Five operators and ten motions give fifty commands, without fifty things to memorise.

Operators		Combined with a motion	
d	delete (cut)	dw	delete to the start of the next word
y	yank (copy)	d\$	delete to the end of the line
c	change: delete, then insert	cG	change to the end of the file
>	indent	y}	yank to the end of the paragraph
<	unindent	c%	change to the matching bracket
g~	swap case	gUiw	uppercase the current word
=	auto-indent	=G	re-indent to the end of the file

Doubling an operator applies it to the whole line: **dd** deletes a line, **yy** yanks one, **>>** indents one.

9.6.1 Text objects

Text objects are Vim's addition to the grammar, and the single feature most worth learning early. Instead of describing *how far* to go, they describe *what structure* to act on — and the cursor can be anywhere inside it.

Each is written as `i` (*inner*, the contents) or `a` (*a*, contents plus the delimiters or trailing space) followed by the object.

Object	Selects	Example	Does
<code>iw aw</code>	a word	<code>diw</code>	delete the word under the cursor
<code>is as</code>	a sentence	<code>das</code>	delete the whole sentence
<code>ip ap</code>	a paragraph	<code>yip</code>	yank the paragraph
<code>i" a"</code>	text in double quotes	<code>ci"</code>	replace what is between the quotes
<code>i' a'</code>	text in single quotes	<code>di'</code>	empty the quoted string
<code>i(a(</code>	text in parentheses	<code>ci(</code>	replace the argument list
<code>i{ a{</code>	text in braces	<code>da{</code>	delete a whole block
<code>i[a[</code>	text in square brackets	<code>yi[</code>	yank an array index
<code>it at</code>	an XML or HTML tag	<code>cit</code>	replace the contents of a tag

Tip

`ci"` is the one to try first. Put the cursor anywhere on the line — inside the quotes, on a quote, it does not matter — and Vim deletes the quoted text and drops you into Insert mode ready to type the replacement. There is no equivalent in a conventional editor that does not involve careful selecting.

9.7 Editing

Key	Effect
<code>x</code>	Delete the character under the cursor.
<code>X</code>	Delete the character before the cursor.
<code>dd</code>	Delete the whole line.
<code>5dd</code>	Delete five lines.
<code>dw</code>	Delete from the cursor to the start of the next word.
<code>D</code>	Delete to the end of the line (same as <code>d\$</code>).
<code>yy</code>	Yank (copy) the current line.
<code>5yy</code>	Yank five lines.
<code>p</code>	Put (paste) after the cursor, or below the current line.
<code>P</code>	Put before the cursor, or above the current line.
<code>r</code>	Replace the single character under the cursor.
<code>cw</code>	Change a word: delete it and enter Insert mode.
<code>cc</code>	Change the whole line.
<code>C</code>	Change to the end of the line.
<code>s</code>	Substitute one character and enter Insert mode.
<code>J</code>	Join the next line onto this one.
<code>~</code>	Swap the case of the character under the cursor.

Key	Effect
>> <<	Indent, unindent the current line.
u	Undo.
Ctrl+R	Redo.
.	Repeat the last change.

Deleting is cutting

`dd` does not discard the line — it moves it into a register, so `p` pastes it back. `dd` followed by `p` on another line is how you move a line. There is no separate cut command because delete already is one.

The dot command

`.` repeats the last change, and it is the reason Vim users make small, repeatable edits. Change one word with `ciw`, type the replacement, press `Esc`; then move to the next occurrence and press `.` — the same change happens again. Combined with `n` to jump between search matches, it is a manual find-and-replace you can steer.

9.8 Multi-level undo

Original `vi` had a single level of undo: `u` undid the last change, and pressing it again *redid* it. Vim keeps a full history.

Command	Effect
u	Undo one change. Repeatable.
Ctrl+R	Redo.
5u	Undo the last five changes.
U	Undo all recent changes on one line.
:earlier 10m	Put the file back as it was ten minutes ago.
:later 5m	Move forward again.
:undolist	Show the branches of the undo tree.

Undo that survives closing the file

Add `set undofile` to your `~/.vimrc` (Section 9.14) and Vim writes the undo history to disk, so you can reopen a file tomorrow and still undo yesterday's edits.

9.9 Visual mode

Visual mode inverts the grammar: instead of “operator, then what to act on”, you select first and then apply the operator. Useful when the region is easier to see than to describe.

Key	Selects
<code>v</code>	Character by character.
<code>V</code>	Whole lines.
<code>Ctrl+V</code>	A rectangular <i>block</i> — columns, not lines.
<code>gv</code>	Reselect whatever was selected last.

Extend the selection with any motion or text object, then press an operator: `d` to delete, `y` to yank, `c` to change, `>` to indent, `u` or `U` to change case.

9.9.1 Block editing

`Ctrl+V` is the feature people usually discover last and then use constantly. It selects a rectangle across several lines, so you can insert or delete the same text in a column.

To comment out five lines of a shell script:

Step	What happens
<code>Ctrl+V</code>	Enter block visual mode at the first line.
<code>4j</code>	Extend the block down four lines.
<code>I</code>	Insert at the left edge of the block.
<code>#</code> Space	Type the comment marker — it appears on the first line only, so far.
Esc	The insertion is replicated onto all five lines.

To remove them again: `Ctrl+V`, `4j`, `ll` to cover both characters, then `d`.

9.10 Registers

Every yank and delete goes into a *register*. Vim has many, not one, so a delete need not destroy what you copied earlier.

Register	Holds
<code>" "</code>	The unnamed register — the most recent yank or delete. What <code>p</code> uses by default.
<code>"0</code>	The most recent <i>yank</i> only. Survives subsequent deletes.
<code>"a ... "z</code>	Named registers you fill deliberately: <code>"a_{yy}</code> yanks a line into register <code>a</code> , <code>"a_p</code> pastes it back.
<code>"A ... "Z</code>	The same registers, but <i>appending</i> instead of replacing.
<code>"+</code>	The system clipboard.
<code>"%</code>	The current file name. Read-only.
<code>"/</code>	The last search pattern. Read-only.
<code>:registers</code>	Show the contents of every register.

The classic frustration, solved

You yank a line, delete another line to make room, then paste — and get the deleted line back instead of the one you copied. The delete overwrote the unnamed register. Paste from the yank register instead: `"0p`.

Clipboard support

`"+y` copies to the system clipboard and `"+p` pastes from it, but only if Vim was built with clipboard support. Check with `vim --version | grep clipboard` — you want `+clipboard`, not `-clipboard`. On Ubuntu the minimal package lacks it; `sudo apt install vim-gtk3` provides it.

9.11 Searching and replacing

Command	Effect
<code>/pattern</code>	Search forwards.
<code>?pattern</code>	Search backwards.
<code>n</code>	Next match, in the same direction.
<code>N</code>	Previous match.
<code>*</code>	Search forwards for the word under the cursor.
<code>#</code>	Search backwards for it.
<code>:noh</code>	Clear the search highlighting.
<code>:s/old/new/</code>	Replace the first occurrence on this line.
<code>:s/old/new/g</code>	Replace all occurrences on this line.
<code>:%s/old/new/g</code>	Replace all occurrences in the whole file.
<code>:%s/old/new/gc</code>	The same, asking for confirmation each time.
<code>:1,20s/old/new/g</code>	Replace only between lines 1 and 20.
<code>:g/pattern/d</code>	Delete every line matching the pattern.
<code>:v/pattern/d</code>	Delete every line <i>not</i> matching it.

Note

`%` means “every line”, `g` means “every occurrence on the line” and `c` means “confirm”. Leave `g` off and only the first match on each line is changed — a common and quiet mistake.

`:g` is a small power tool

`:g/pattern/command` runs an Ex command on every matching line. `:g/DEBUG/d` strips every debug line from a log; `:g/^$/d` removes blank lines; `:g/TODO/t$` copies every TODO line to the end of the file.

9.12 Macros

A *macro* records a sequence of keystrokes into a register and replays it. It is the right tool when an edit is too irregular for a substitution but has to be repeated many times.

Key	Effect
qa	Start recording into register <i>a</i> . Vim shows <code>recording @a</code> .
q	Stop recording.
@a	Replay the macro once.
20@a	Replay it twenty times.
@@	Replay the most recently used macro.

A worked example — turning a plain list of names into a quoted, comma-separated list:

```
Saman      ->  "Saman",
Kumar      "Kumar",
Kapila     "Kapila",
```

Keys	What they do
qa	Start recording into register <i>a</i> .
I " Esc	Insert a quote at the start of the line.
A " , Esc	Append a quote and a comma at the end.
j	Move down to the next line — essential, or the replay will keep editing the same line.
q	Stop recording.
10@a	Apply it to the next ten lines.

Tip

End every macro with the motion that sets up the next repetition, and start it from a predictable position (`0` or `I`). A macro that assumes the cursor is already in the right column works once and then corrupts the rest of the file.

9.13 Buffers, windows and tabs

Vim distinguishes three things that other editors merge into one.

Term	Is
Buffer	A file loaded into memory. Open ten files and you have ten buffers, whether or not you can see them.
Window	A viewport onto a buffer. Two windows can show the same buffer, or different ones.
Tab	A named layout of windows — not a file, as it is in most editors.



Figure 9.3. Two windows can show the same buffer, and a buffer can be open with no window at all.

Command	Effect
<code>:ls</code> or <code>:buffers</code>	List the open buffers.
<code>:b 2</code>	Switch to buffer 2. <code>:b name</code> also works.
<code>:bn</code> <code>:bp</code>	Next, previous buffer.
<code>:bd</code>	Close the current buffer.
<code>:sp file</code>	Split horizontally and open a file.
<code>:vsp file</code>	Split vertically.
<code>Ctrl+W h j k l</code>	Move between windows.
<code>Ctrl+W q</code>	Close the current window.
<code>Ctrl+W =</code>	Make all windows the same size.
<code>:tabnew file</code>	Open a file in a new tab.
<code>gt</code> <code>gT</code>	Next, previous tab.
<code>:tabclose</code>	Close the current tab.

Tip

`:vsp` plus `Ctrl+W l` is how you read a reference file while writing — a script on the left, its output or a manual page on the right, both in one terminal over one SSH connection.

9.14 Configuring Vim

Settings typed with a colon apply to the current session only. To make them permanent, put them — without the colon — in `~/.vimrc`.

Setting	Effect
<code>:set number</code>	Show line numbers. <code>:set nonumber</code> hides them.
<code>:set relativenumber</code>	Number lines relative to the cursor, which makes counts like <code>7j</code> easy to judge.
<code>:syntax on</code>	Syntax highlighting.
<code>:set ignorecase</code>	Case-insensitive searching.
<code>:set smartcase</code>	... unless the pattern contains a capital.

Setting	Effect
<code>:set incsearch</code>	Jump to matches as you type the pattern.
<code>:set hlsearch</code>	Highlight every match.
<code>:set autoindent</code>	Keep the previous line's indentation.
<code>:set expandtab</code>	Insert spaces rather than tab characters.
<code>:set tabstop=4</code>	Display a tab as four columns.
<code>:set shiftwidth=4</code>	Indent by four with <code>>></code> .
<code>:set undofile</code>	Keep undo history between sessions.
<code>:set paste</code>	Stop auto-indent mangling pasted text.
<code>:set spell</code>	Spell checking; <code>z=</code> suggests corrections.
<code>:set nocompatible</code>	Enable Vim's improvements over <code>vi</code> .

A reasonable starting `~/.vimrc`:

```
cat > ~/.vimrc <<'EOF'
set nocompatible          " behave as Vim, not as vi
syntax on                 " colour by language
set number                " line numbers
set autoindent            " keep the previous indent
set tabstop=4             " a tab shows as four columns
set shiftwidth=4          " >> indents by four
set expandtab              " insert spaces, not tab characters
set ignorecase smartcase  " smart case-insensitive search
set incsearch hlsearch    " search as you type, highlight matches
set undofile              " undo history survives closing the file
set wildmenu              " tab completion on the : command line
set backspace=indent,eol,start
EOF
```

Note

In a `.vimrc`, a double quote starts a comment — not `#`, as in a shell script. This catches everyone once.

9.15 Plugins

Vim 8 and later can load plugins with no extra tooling: drop them into `~/.vim/pack/plugins/start/` and they load at startup.

```
mkdir -p ~/.vim/pack/plugins/start
cd ~/.vim/pack/plugins/start
git clone https://github.com/tpope/vim-surround
```

That single command gives you `cs"` to change surrounding double quotes to single ones, and `ysiw` to wrap a word in quotes — a natural extension of the grammar in Section 9.6.

Plugin	Adds
<code>vim-surround</code>	Operators for the brackets and quotes around text.
<code>vim-commentary</code>	<code>gc</code> to comment a motion or selection.
<code>vim-fugitive</code>	Git from inside Vim (Chapter 13).
<code>nerdtree</code>	A file browser in a side window.
<code>vim-airline</code>	An informative status line.

Caution

Add plugins one at a time, and only when you have felt the need the plugin answers. A large borrowed `.vimrc` makes Vim behave in ways you cannot explain or fix, which is worse than a plain install.

9.16 A first session

The shortest path from nothing to a saved file:

Step	What happens
<code>vim notes.txt</code>	Opens an empty buffer. You are in Normal mode.
<code>i</code>	Insert mode. The status line shows <code>-- INSERT --</code> .
type your text	Characters appear as expected.
<code>Esc</code>	Back to Normal mode. The status line clears.
<code>:wq</code>	Write the file and quit.

If Vim seems frozen

Most likely you are in Normal mode and your keystrokes are being interpreted as commands. Press `Esc`, then `u` repeatedly to undo any damage. If the terminal itself has stopped responding, you may have pressed `Ctrl+S`, which pauses output — `Ctrl+Q` resumes it. And if you cannot work out how to leave, `Esc` then `:q!` always exits without saving.

Going further

Run `vimtutor` from the command line. It is a thirty-minute interactive tutorial that ships with Vim and covers the first half of this chapter hands-on, in a throwaway copy you cannot damage. After that, the fastest way to improve is to notice a repetitive edit *while* you are making it and ask whether an operator, a text object or a macro would do it in one command — `:help` plus a keyword answers most such questions offline. Two directions open up from there: *Neovim*, a compatible fork with a built-in terminal, Lua configuration and language-server support; and Vim keybindings in other tools — VS Code, JetBrains IDEs, `less`, `bash` (`set -o vi`) and your browser all offer them, so the investment is not confined to one editor.

CHAPTER 10

Basic System Administration

Shutting the machine down properly, understanding what state it boots into, installing software, packing files up for transfer, watching processes and checking the network. The everyday administration a user of a Linux system needs.

10.1 Shutting down and rebooting

Never switch a Linux machine off at the power button. Writes are buffered in memory, services need to close their files, and other users may be logged in.

Command	Effect
<code>shutdown -h now</code>	Halt immediately.
<code>shutdown -r now</code>	Reboot immediately.
<code>shutdown -h +10</code>	Halt in ten minutes, warning logged-in users.
<code>shutdown -h 22:00</code>	Halt at ten o'clock.
<code>shutdown -c</code>	Cancel a scheduled shutdown.
<code>poweroff</code>	Halt immediately. Available on most systems.
<code>reboot</code>	Reboot immediately.
<code>halt</code>	Stop the system without powering it off.
<code>systemctl poweroff</code>	The systemd equivalents, on modern systems.
<code>systemctl reboot</code>	

All of these require root, so in practice: `sudo shutdown -h now`.

10.2 Runlevels and targets

A *runlevel* is a mode of operation in operating systems implementing UNIX System V-style initialisation. It defines the state of the machine after boot — in effect, how many of the services and daemons are running.

Conventionally seven runlevels exist, numbered zero to six. `s` is sometimes used as a synonym for one of them. Only one runlevel is executed at startup: they are not executed one after another, so runlevel 2, 3 or 5 runs, not several in sequence.

Level	Meaning	systemd target
0	Halt	<code>poweroff.target</code>
1	Single user (maintenance)	<code>rescue.target</code>
2	Multi-user, no network (traditionally)	<code>multi-user.target</code>
3	Full multi-user, text console	<code>multi-user.target</code>
4	Unused / site-defined	<code>multi-user.target</code>
5	Multi-user with a graphical login	<code>graphical.target</code>
6	Reboot	<code>reboot.target</code>

Distributions differ

The meanings above are conventions, not guarantees. Slackware, for example, uses runlevel 1 for maintenance as others do, but configures runlevels 2, 3 and 5 identically for a console with all services active, and adds the X Window System at runlevel 4 — the reverse of the Red Hat convention.

10.2.1 systemd

Modern distributions have replaced the runlevel system with `systemd`, which uses named *targets* instead of numbers. The old commands are kept as compatibility aliases.

Command	Effect
<code>systemctl get-default</code>	Show the target the system boots into.
<code>systemctl set-default multi-user.target</code>	Boot to a text console in future.
<code>systemctl isolate graphical.target</code>	Switch to that target now.
<code>runlevel</code>	Show the previous and current runlevel.

10.2.2 Managing services

Command	Effect
<code>systemctl status ssh</code>	Is it running, and what has it logged recently?
<code>systemctl start ssh</code>	Start it now.
<code>systemctl stop ssh</code>	Stop it now.
<code>systemctl restart ssh</code>	Stop and start — needed after a config change.
<code>systemctl reload ssh</code>	Re-read the configuration without dropping connections, where the service supports it.
<code>systemctl enable ssh</code>	Start automatically at boot.
<code>systemctl disable ssh</code>	Do not start at boot.
<code>systemctl is-enabled ssh</code>	Report whether it is enabled.
<code>journalctl -u ssh</code>	Show that service's log. <code>-f</code> follows it live.

Note

`start` and `enable` are independent. `start` affects the machine now; `enable` affects what happens at the next boot. A service you started but did not enable disappears after a reboot — a common source of confusion when setting up SSH in Chapter 15.

10.3 Installing software

Linux installs software from *repositories* through a *package manager*, rather than by downloading installers. The package manager resolves dependencies, verifies signatures and keeps track of what is installed.

Task	Debian / Ubuntu	Red Hat / Fedora
Refresh the package list	<code>apt update</code>	<code>dnf check-update</code>
Upgrade everything	<code>apt upgrade</code>	<code>dnf upgrade</code>
Install	<code>apt install vim</code>	<code>dnf install vim</code>
Remove	<code>apt remove vim</code>	<code>dnf remove vim</code>
Search	<code>apt search editor</code>	<code>dnf search editor</code>
Show details	<code>apt show vim</code>	<code>dnf info vim</code>
List installed	<code>dpkg -l</code>	<code>rpm -qa</code>
Which package owns a file	<code>dpkg -S /bin/ls</code>	<code>rpm -qf /bin/ls</code>
Install a local package	<code>dpkg -i pkg.deb</code>	<code>rpm -i pkg.rpm</code>

```
sudo apt update                # always refresh first
sudo apt install openssh-server # then install
```

Note

`rpm` stands for *Red Hat Package Manager* and names both the tool and the `.rpm` file format. `dpkg` is its Debian counterpart, handling `.deb` files. Both are low-level: they install a single package and report missing dependencies but will not fetch them. `apt` and `dnf` sit on top and do the fetching, which is why you almost always use those instead.

10.4 Packing and unpacking

The equivalent of WinZip or WinRAR on Windows, but split into two jobs: `tar` bundles many files into one, and a separate compressor shrinks it.

`tar` stands for *tape archive* — it was written to write files to magnetic tape, which is also why its options look unlike anything else.

Option	Meaning
c	Create a new archive.
x	Extract from an archive.
t	List (test) the contents without extracting.
v	Verbose — print each file name as it is processed.
f	The next argument is the archive <i>file</i> name. Nearly always needed, and must come last in a combined group.
z	Compress or decompress with gzip (<i>.tar.gz</i> , <i>.tgz</i>).
j	The same with bzip2 (<i>.tar.bz2</i>).
J	The same with xz (<i>.tar.xz</i>).

```

tar cvf archive.tar mydir/      # create an uncompressed archive
tar cvzf archive.tar.gz mydir/  # create and gzip it
tar tf archive.tar              # list the contents
tar tzf archive.tar.gz          # list the contents of a gzipped archive
tar xvf archive.tar             # extract
tar xvzf archive.tar.gz         # extract a gzipped archive
tar xvzf archive.tar.gz -C /tmp # extract somewhere specific

# the common submission pattern: collect, then compress
mkdir Myname_IndexNo
cp -r answers/* Myname_IndexNo/
tar cvzf Myname_IndexNo.tar.gz Myname_IndexNo/

```

Other tools	Notes
gzip file.txt	Compress in place, producing <i>file.txt.gz</i> and removing the original.
gunzip file.txt.gz	Decompress it again.
zip -r a.zip dir/	Create a ZIP archive — use this when the recipient is on Windows.
unzip a.zip	Extract a ZIP archive.

Why **f** is conventionally written last

Each option letter that takes a value consumes the next command-line argument, in the order the letters appear. Only **f** does, so GNU tar accepts `tar cvzf` and `tar cvfz` alike. Keeping **f** last is still the right habit: it is required by some non-GNU implementations, and it puts the archive name immediately after the letter that claims it. The long forms — `--create --gzip --file=archive.tar.gz` — remove the ambiguity entirely.

10.5 Processes

Command	Shows / does
<code>ps</code>	Processes started from this terminal.
<code>ps aux</code>	Every process on the system, with owner and resource use.
<code>ps aux grep firefox</code>	Find a specific process.
<code>top</code>	A live, continuously updating process table. <code>q</code> quits.
<code>htop</code>	A friendlier version, if installed.
<code>kill 1234</code>	Ask process 1234 to terminate cleanly.
<code>kill -9 1234</code>	Force it to stop. A last resort — it cannot clean up after itself.
<code>killall firefox</code>	Signal every process with that name.
<code>pgrep sshd</code>	Find process IDs by name.

10.6 Disk and system information

Command	Reports
<code>df -h</code>	Free space on each mounted filesystem.
<code>du -sh *</code>	Size of each item in the current directory.
<code>free -h</code>	Memory and swap in use.
<code>uname -a</code>	Kernel name, version, architecture — everything.
<code>uname -r</code>	Just the kernel release.
<code>hostname</code>	The machine's name.
<code>uptime</code>	How long the system has been up, and its load average.
<code>date</code>	Current date and time.
<code>whoami</code>	Current username.
<code>cat /etc/os-release</code>	Distribution name and version.
<code>lscpu</code>	Processor details.
<code>lsblk</code>	Block devices and their mount points.

10.7 Networking

Command	Purpose
<code>ip a</code>	Show network interfaces and their addresses. The modern replacement for <code>ifconfig</code> .
<code>ip r</code>	Show the routing table.
<code>ifconfig</code>	The older equivalent of <code>ip a</code> ; may not be installed by default.
<code>ping google.com</code>	Test whether a host is reachable. Ctrl+C to stop.
<code>ping -c 4 192.168.1.1</code>	Send exactly four packets.
<code>traceroute google.com</code>	Show the route packets take.
<code>ss -tuln</code>	List listening TCP and UDP ports. The modern replacement for <code>netstat</code> .
<code>netstat -tuln</code>	The older equivalent.
<code>ss -tuln grep :22</code>	Check whether anything is listening on port 22.
<code>curl -I https://site.com</code>	Fetch just the HTTP headers from a URL.
<code>wget https://site.com/f.zip</code>	Download a file.
<code>dig example.com</code>	Query DNS.

Tip

`ip a` is how you find the address to connect to when setting up SSH in Chapter 15. Look for an interface other than `lo` (the loopback) and read the address after `inet`.

10.7.1 The firewall

Command	Effect
<code>sudo ufw status</code>	Show the firewall rules (Ubuntu).
<code>sudo ufw allow 22</code>	Allow inbound connections on port 22.
<code>sudo ufw allow ssh</code>	The same, by service name.
<code>sudo ufw reload</code>	Re-apply the rules.
<code>sudo ufw enable</code>	Turn the firewall on.

Caution

Enabling a firewall on a remote machine before allowing SSH will lock you out of it. Always `sudo ufw allow 22` *first*.

10.8 Changing your password

```
passwd          # change your own; asks for the current one first
sudo passwd bob # change another user's, as root
```

Nothing is echoed as you type — not even asterisks. This is deliberate: it conceals the length of the password from anyone watching.

Going further

Three areas open up from here. *Logs*: `journalctl` and the files under `/var/log` are where you look first when something has failed; `journalctl -xe` shows the most recent errors with context. *Scheduling*: `cron` runs commands on a timetable, and `crontab -e` edits yours — combined with Chapter 12 this is how routine work gets automated. *Configuration management*: once you are setting up more than one machine, tools such as Ansible describe the desired state in a file instead of you typing the same commands twice.

PART III

The Bash Shell

CHAPTER 11

The Bash Shell

Bash is not just a place to type commands — it is a programming language whose interactive mode you have been using all along. This chapter covers what the shell does to a command line before the command runs: expansion, quoting, redirection and pipes. Chapter 12 then puts the same language in a file.

11.1 What the shell is

Bash — the *Bourne Again Shell* — is the default on most Linux distributions. Its job is to read a line, transform it according to a fixed set of rules, find the program you named, and run it with the resulting arguments.

Almost every surprise at the command line comes from the transformation step. The command never sees what you typed; it sees what the shell made of it.

11.1.1 Startup files

File	Read when
<code>/etc/profile</code>	Any login shell starts. System-wide.
<code>~/.bash_profile</code>	A login shell starts — when you log in at a console or over SSH.
<code>~/.bashrc</code>	An <i>interactive non-login</i> shell starts — when you open a new terminal window. This is where aliases and prompt settings belong.
<code>~/.bash_logout</code>	A login shell exits.

Changes to these files take effect at the next shell. To apply them now:

```
source ~/.bashrc    # read the file into the current shell
. ~/.bashrc         # the same thing, written the short way
```

Note

`source` is not the same as running the script. Running it starts a new shell, which exits and takes any changes with it. `source` executes the lines in the *current* shell, which is the only way a script can change your environment.

11.2 Variables

```
name="Application Laboratory"  # no spaces around the = sign
echo $name                    # use it with a leading $
echo ${name}                  # braces: needed when text follows
```

```
echo "${name}_2026" # without braces this reads $name_2026
```

Caution

`name = "value"` with spaces does not assign anything — the shell tries to run a command called `name` with the arguments `=` and `"value"`. This is the single most common Bash error.

11.2.1 Environment variables

A shell variable is private to the shell that set it. `export` promotes it to an *environment variable*, which every program the shell starts will inherit.

```
export EDITOR=vim # child processes will see EDITOR
env # list every environment variable
echo $PATH # where the shell searches for commands
unset name # remove a variable
```

Variable	Holds
\$HOME	Your home directory.
\$USER	Your username.
\$PWD	The current working directory.
\$PATH	Colon-separated list of directories searched for commands.
\$SHELL	The path of your login shell.
\$?	Exit status of the last command. Zero means success.
\$\$	Process ID of the current shell.
\$RANDOM	A different random number between 0 and 32767 each time it is read.

11.3 Quoting

Quoting controls how much of the shell's transformation applies to a piece of text. Three levels, and the difference matters constantly.

Form	Protects against	Still happens inside
No quotes	nothing	Everything: expansion, word splitting, globbing.
"double"	word splitting and globbing	Variable expansion, command substitution, and <code>\</code> escapes.
'single'	everything	Nothing at all — the text is entirely literal.
<code>\c</code>	the single character <code>c</code>	—

```
name="Application Lab"

echo $name # Application Lab -- two words, then rejoined by echo
echo "$name" # Application Lab -- one word: what you almost always
    ↪ want
echo '$name' # $name -- literal, no expansion
```

```

echo "Path: $HOME" # Path: /home/student
echo 'Path: $HOME' # Path: $HOME

touch "my file.txt" # one file with a space in its name
touch my\ file.txt # the same thing, escaped

```

The rule worth memorising

Put double quotes around every variable expansion unless you have a specific reason not to. `rm $file` breaks when the filename contains a space; `rm "$file"` does not.

11.4 echo and escape sequences

Option	Effect
<code>-e</code>	Interpret backslash escapes in the argument.
<code>-n</code>	Do not print a trailing newline.

<code>\n</code>	newline	<code>\t</code>	tab
<code>\\</code>	a literal backslash	<code>\"</code>	a literal double quote

```

echo -e "Bash\nCopy\nEdit" # three lines
echo "Linux\\Training"      # Linux\Training
echo 'Linux\Training'       # the same, more simply
echo "This is quoted text"  # quotes are consumed by the shell
echo '"This is quoted text"' # to print the quotes, quote them
echo "4+4 =8"
printf "%s\t%s\n" "Name" "Score" # printf, when the layout matters

```

Note

`printf` is more predictable than `echo` for anything beyond a plain message: `echo`'s handling of options and escapes varies between shells, while `printf` behaves the same everywhere.

11.5 Expansion

Before running anything, Bash rewrites the line in a fixed order.

what you type

```
echo {1..3} ~/*.txt $USER $(date +%Y) $((2*21))
```

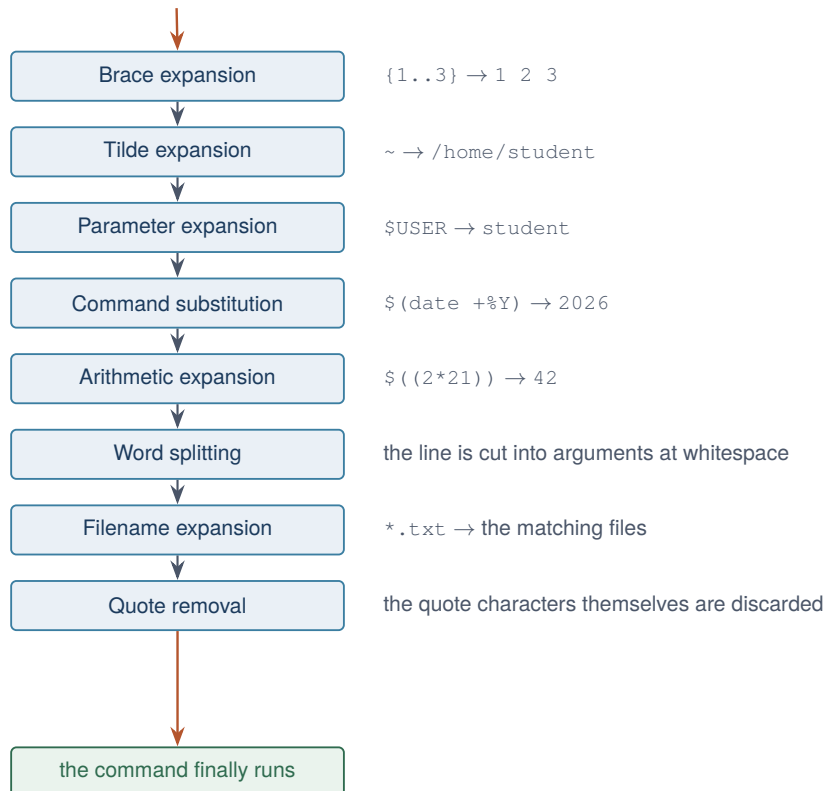


Figure 11.1. Bash rewrites the command line in this order. Because filename expansion happens near the end, a variable holding `*` will still be globbed unless it is quoted.

11.5.1 Brace expansion

Generates a list of strings. It happens first, and it does not care whether the resulting names exist.

```
echo {1..5}           # 1 2 3 4 5
echo {a..e}           # a b c d e
echo {1..10..2}       # 1 3 5 7 9
echo file{1,2,3}.txt  # file1.txt file2.txt file3.txt
mkdir -p WebApp/{HTML,Assets,Logs}
touch student{1..8}.txt # eight files in one command
touch invoice_Q{1..4}.txt # invoice_Q1.txt ... invoice_Q4.txt
cp report.txt report.txt.{bak,orig}
```

11.5.2 Command substitution

Runs a command and replaces the whole construct with its output.

```
today=$(date +%Y-%m-%d)
echo "Report generated on $today"

host=$(hostname)
user=$(whoami)
echo "Prepared by $user on $host"
```

```
count=$(ls *.txt | wc -l)
echo "There are $count text files"
```

Note

The older syntax uses backticks: ``date``. Prefer `$ ( )` — it nests without confusion and is far easier to read.

11.5.3 Arithmetic expansion

```
echo $((2 + 3))          # 5
echo $((10 / 3))         # 3  -- integer division only
echo $((10 % 3))         # 1  -- remainder
n=5; echo $((n * 12))    # 60 -- no $ needed inside (( ))

mark=$(( RANDOM % 61 + 40 )) # a random integer from 40 to 100
```

Caution

Bash arithmetic is integer-only: `$((7/2))` is 3, not 3.5. For decimals, pipe through `bc`: `echo "scale=2; 7/2" | bc`.

11.6 Aliases

An *alias* is a short name for a longer command.

Command	Effect
<code>alias</code>	List every alias currently defined.
<code>alias hello='echo "Hello from Ravindu!"'</code>	Define one.
<code>alias ll='ls -alF'</code>	A typical one.
<code>alias grep rm</code>	Find whether <code>rm</code> has an alias.
<code>type rm</code>	The same question, answered more fully.
<code>unalias rm</code>	Remove it, for this session only.
<code>\rm file</code>	Bypass an alias once, without removing it.

```
alias hello='echo "Hello from Ravindu!"'
hello                                # test it
alias                                # list all aliases; look for one on rm
unalias rm                           # remove it temporarily
```

Aliases defined at the prompt vanish when the shell exits. To keep one, add the `alias` line to `~/.bashrc`.

Note

Many distributions ship `alias rm='rm -i'` so that deletion always prompts. It is a useful safety net — but it means `rm` behaves differently on your machine than in a script, which is why removing it is a standard exercise.

11.7 Redirection

Every process starts with three open streams. Redirection reconnects them.

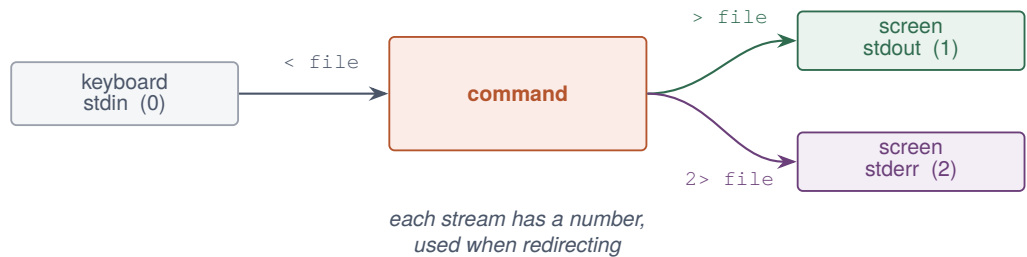


Figure 11.2. Standard input, standard output and standard error. Errors travel on a separate stream so that they stay visible even when output is being captured.

Form	Effect
cmd > file	Send output to the file, replacing its contents.
cmd >> file	Append output to the file.
cmd < file	Take input from the file instead of the keyboard.
cmd 2> file	Send error messages to the file.
cmd 2>> file	Append error messages.
cmd > out 2> err	Separate the two streams into two files.
cmd > file 2>&1	Send both to the same file.
cmd &> file	The same, written more briefly (Bash only).
cmd 2> /dev/null	Discard error messages.
cmd > /dev/null	Discard everything.
2>&1	
cmd <<EOF ... EOF	Here-document: feed the following lines in as input.

```
ls -l > permissions_check.txt      # save a listing
echo "line one" > report.txt       # create with one line
echo "line two" >> report.txt       # add another
find / -name "*.conf" 2> /dev/null # hide the permission-denied noise
./script.sh > output.log 2>&1      # capture everything a script produces
```

Caution

The order in `> file 2>&1` matters. It reads “send stdout to the file, then send stderr wherever stdout is now going”. Reversed — `2>&1 > file` — stderr is pointed at the screen *before* stdout is redirected, so errors still appear on the terminal.

11.8 Pipes

A *pipe* connects one command’s standard output to the next command’s standard input, with no temporary file in between.



each command's output becomes the next one's input

Figure 11.3. `ls -l | grep .txt | wc -l` — list files, keep the text files, count them.

```

ls -l | less           # page through a long listing
ps aux | grep firefox # find a process
cat access.log | grep "404" | wc -l # count 404s in a log
du -sh * | sort -h     # rank items in this directory by size
history | grep chmod   # what chmod commands have I run?
find . -type f | wc -l # how many files below here?
cat file.txt | tee copy.txt | wc -l # tee: save a copy and pass it on
  
```

Pipe versus redirect

> sends output to a *file*. | sends output to another *command*. `ls > grep` creates a file called `grep` — a mistake that is easy to make and confusing to diagnose.

11.8.1 Filters worth knowing

Commands designed to sit in the middle of a pipeline.

Command	Does
<code>grep pattern</code>	Keep only the lines that match.
<code>grep -i pattern</code>	Case-insensitive.
<code>grep -v pattern</code>	Invert: keep the lines that do <i>not</i> match.
<code>grep -c pattern</code>	Count matching lines instead of printing them.
<code>grep -l pattern *</code>	Print the <i>names</i> of files that contain a match.
<code>grep -r pattern dir</code>	Search recursively through a directory.
<code>grep -n pattern</code>	Prefix each match with its line number.
<code>sort</code>	Sort lines alphabetically. <code>-n</code> numerically, <code>-r</code> reversed, <code>-h</code> human-readable sizes.
<code>uniq</code>	Collapse adjacent duplicate lines. <code>-c</code> counts them. Sort first.
<code>wc -l</code>	Count lines. <code>-w</code> words, <code>-c</code> bytes.
<code>head -n 5</code>	The first five lines.
<code>tail -n 5</code>	The last five.
<code>cut -d: -f1</code>	Field 1 of each line, using <code>:</code> as the separator.
<code>tr 'a-z' 'A-Z'</code>	Translate characters — here, to uppercase.
<code>sed 's/old/new/g'</code>	Substitute text as the lines stream past.
<code>awk '{print \$2}'</code>	Print the second whitespace-separated field.
<code>tee file</code>	Write the stream to a file <i>and</i> pass it on.
<code>xargs rm</code>	Turn incoming lines into arguments for a command.

```
cut -d: -f1 /etc/passwd | sort           # every username, alphabetically
grep -l "3" student*.txt                # which files contain a 3
grep -c "error" server.log              # how many error lines
sort marks.txt | uniq -c | sort -rn      # frequency table, most common first
```

11.9 Job control

Command	Effect
<code>cmd &</code>	Run in the background; the prompt returns at once.
Ctrl+Z	Suspend the running job.
<code>jobs</code>	List the shell's jobs.
<code>bg</code>	Resume the suspended job in the background.
<code>fg</code>	Bring the most recent background job to the foreground.
<code>fg %2</code>	Bring job number 2 forward.
Ctrl+C	Terminate the foreground job.
<code>nohup cmd &</code>	Run it so that it survives your logging out.

11.10 Debugging what the shell does

`set -x` makes Bash print each command after expansion, immediately before running it. This is how you see what the shell actually built.

```
$ set -x
$ echo "Hello"
+ echo Hello
Hello
$ echo $HOME
+ echo /home/student
/home/student
$ set +x
+ set +x
```

Command	Effect
<code>set -x</code>	Turn tracing on. Traced lines are prefixed with +.
<code>set +x</code>	Turn tracing off.
<code>set -e</code>	Exit immediately if any command fails.
<code>set -u</code>	Treat the use of an unset variable as an error.
<code>set -o pipefail</code>	A pipeline fails if <i>any</i> stage fails, not just the last.

Tip

– turns a shell option *on* and + turns it *off* — the opposite of what the symbols suggest. `set -euo pipefail` at the top of a script is a widely used combination; see Chapter 12.

11.11 History

Command	Effect
<code>history</code>	List previous commands, numbered.
<code>history grep chmod</code>	Find one you ran earlier.
<code>!42</code>	Re-run command number 42.
<code>!!</code>	Re-run the previous command.
<code>sudo !!</code>	Re-run it with <code>sudo</code> — the classic use.
<code>!ls</code>	Re-run the most recent command starting with <code>ls</code> .
<code>Ctrl+R</code>	Search history interactively as you type.

History is kept in `~/.bash_history` and written when the shell exits.

Going further

Two habits pay off immediately. Use `Ctrl+R` instead of pressing `↑` repeatedly — it finds any command you have run, however long ago. And when a pipeline is not doing what you expect, build it up one stage at a time, checking the output at each step, rather than debugging the whole thing at once. For depth, `man bash` is the complete reference, and the *Bash Guide* at <https://mywiki.woledge.org/BashGuide> is the best free tutorial. `grep`, `sed` and `awk` each repay separate study; regular expressions underpin all three.

CHAPTER 12

Bash Scripting

A script is a file of shell commands. Everything in Chapter 11 still applies; this chapter adds the constructs that turn a list of commands into a program — variables, conditionals, loops and functions — and closes with worked scripts built from the laboratory tasks.

12.1 Anatomy of a script

```
#!/bin/bash
# setup.sh -- create the working directory for today's session
# Author: <your name>

set -euo pipefail

TARGET="$HOME/student_records"
mkdir -p "$TARGET"
echo "Created $TARGET"
```

Line	Purpose
#!/bin/bash	The <i>shebang</i> . Tells the kernel which interpreter should run this file. It must be the very first line, with no blank line or space before it.
# comment	Everything after # is ignored. Say <i>why</i> , not <i>what</i> .
set -euo pipefail	Fail fast: stop on the first error, on any unset variable, and on a failure anywhere in a pipeline.

12.1.1 Making it run

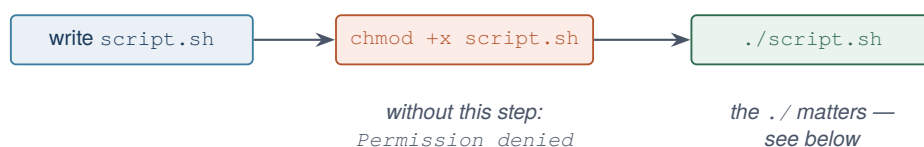


Figure 12.1. A new file is not executable. Bash also will not search the current directory for commands, so the path must be explicit.

Command	Effect
<code>./script.sh</code>	Run it as a program. Requires the execute bit and the shebang. Runs in a <i>new</i> shell.
<code>bash script.sh</code>	Run it by naming the interpreter. Needs no execute bit, and ignores the shebang.
<code>source script.sh</code>	Run the lines in the <i>current</i> shell, so any variables or directory changes persist afterwards.
<code>. script.sh</code>	The same as <code>source</code> .

Why ./

The current directory is deliberately not in `$PATH`. If it were, someone could leave a malicious file named `ls` in a shared directory and wait for you to type `ls` there. Writing `./script.sh` states plainly that you mean the file in front of you.

12.2 Variables in scripts

```
#!/bin/bash
name="Application Laboratory"      # no spaces around =
count=8
readonly BASE="$HOME/records"     # cannot be reassigned

echo "$name has $count sessions"
echo "Files go in $BASE"
```

12.2.1 Capturing system values

```
current_date=$(date)              # full date and time
today=$(date +%Y-%m-%d)           # just the date, formatted
host=$(hostname)
user=$(whoami)

echo "Date:      $current_date"
echo "Hostname: $host"
echo "User:      $user"
```

12.2.2 Reading input

```
read -p "Enter your name: " username
echo "Hello, $username"

read -s -p "Password: " pass      # -s hides what is typed
echo                             # move to a new line afterwards
```

12.3 Arguments

Values given on the command line arrive in numbered variables.

Variable	Holds
\$0	The name of the script itself.
\$1 ... \$9	The first through ninth arguments.
\${10}	The tenth and beyond — braces are required.
\$#	How many arguments were given.
\$@	All arguments, each as a separate word when quoted: "\$@".
\$*	All arguments as a single word when quoted.
\$?	Exit status of the last command.

```
#!/bin/bash
if [ $# -lt 1 ]; then
    echo "Usage: $0 <directory>"
    exit 1
fi

target="$1"
echo "Working on $target"
```

Tip

Always use "\$@" rather than \$* when passing arguments on to another command. "\$@" preserves the boundaries between arguments, so filenames containing spaces survive.

12.4 Conditionals

12.4.1 Test expressions

The condition of an `if` is a *command*, and its exit status decides the branch. `[ ]` and `[[ ]]` are the usual test commands.

Form	Notes
<code>[...]</code>	POSIX. Portable to any shell. Spaces inside the brackets are mandatory.
<code>[[...]]</code>	Bash. Safer with unquoted variables, and supports pattern matching and <code>&&</code> / <code> </code> inside. Prefer it in Bash scripts.
<code>((...))</code>	Arithmetic evaluation. Use ordinary <code><</code> , <code>></code> and <code>==</code> for numbers.

Numeric comparison —

<code>-eq</code>	equal to	<code>-ne</code>	not equal to
<code>-lt</code>	less than	<code>-le</code>	less than or equal to
<code>-gt</code>	greater than	<code>-ge</code>	greater than or equal to

String comparison —

<code>= or ==</code>	the strings are identical
<code>!=</code>	the strings differ
<code>-z "\$s"</code>	the string is empty
<code>-n "\$s"</code>	the string is not empty

File tests —

<code>-e</code>	exists	<code>-f</code>	exists and is a regular file
<code>-d</code>	is a directory	<code>-s</code>	exists and is not empty
<code>-r</code>	is readable	<code>-w</code>	is writable
<code>-x</code>	is executable	<code>-L</code>	is a symbolic link

Caution

The spaces inside the brackets are part of the syntax. `[$x -eq 1]` fails, because `[` is a command and needs its arguments separated. Write `[ "$x" -eq 1 ]`.

12.4.2 if

```
if [ -d "$HOME/records" ]; then
    echo "Directory already exists"
elif [ -f "$HOME/records" ]; then
    echo "A file of that name is in the way"
else
    mkdir "$HOME/records"
    echo "Created it"
fi

# numeric
if (( score >= 50 )); then
    echo "Pass"
else
    echo "Fail"
fi

# string, with pattern matching
if [[ "$filename" == invoice_Q*.txt ]]; then
    echo "This is an invoice"
fi
```

12.4.3 case

Cleaner than a long `if-elif` chain when testing one value against several patterns.

```
case "$1" in
    start)
        echo "Starting..."
        ;;
    stop)
        echo "Stopping..."
        ;;
    restart|reload)
        echo "Restarting..."
        ;;
    *)
        echo "Unknown command"
        ;;
esac
```

```

        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
        ;;
esac

```

12.5 Loops

12.5.1 for

```

# over a literal list
for colour in red green blue; do
    echo "$colour"
done

# over a numeric range
for i in {1..8}; do
    echo "student$i.txt"
done

# over files -- always quote the variable
for f in *.txt; do
    echo "Processing $f"
done

# C-style
for (( i = 1; i <= 8; i++ )); do
    echo "Iteration $i"
done

# over command output
for user in $(cut -d: -f1 /etc/passwd); do
    echo "$user"
done

```

12.5.2 while and until

```

count=1
while [ $count -le 5 ]; do
    echo "Count: $count"
    (( count++ ))
done

until [ $count -eq 0 ]; do
    (( count-- ))
done

# read a file line by line -- the correct idiom
while IFS= read -r line; do
    echo "Line: $line"
done < input.txt

```

Note

In that last loop, `IFS=` stops leading and trailing whitespace being stripped, and `-r` stops backslashes being interpreted. Without both, the loop silently mangles some inputs.

12.5.3 break and continue

```
for f in *.log; do
    [ -s "$f" ] || continue      # skip empty files
    grep -q "FATAL" "$f" && break # stop at the first fatal error
    echo "Checked $f"
done
```

12.6 Arrays

```
quarters=("Q1" "Q2" "Q3" "Q4")

echo "${quarters[0]}"      # Q1 -- indices start at zero
echo "${quarters[@]}"      # every element
echo "${#quarters[@]}"    # how many elements

for q in "${quarters[@]"; do
    echo "This is the $q invoice summary" > "invoice_$q.txt"
done
```

12.7 Functions

```
#!/bin/bash

# define before use
log() {
    echo "[$(date +%H:%M:%S)] $*"
}

create_record() {
    local file="$1"          # local: invisible outside the function
    local owner="$2"
    {
        echo "Date:      $(date) "
        echo "Hostname: $(hostname) "
        echo "User:      $(whoami) "
        echo "Maintained by $owner"
    } > "$file"
    return 0
}

log "Starting"
create_record "student1.txt" "Ravindu"
log "Done"
```

Feature	Notes
Arguments	Arrive as <code>\$1</code> , <code>\$2</code> ..., exactly as in a script. There is no parameter list in the definition.
<code>local</code>	Confines a variable to the function. Without it, every variable is global — a frequent source of bugs.
<code>return</code>	Sets the exit status (0–255 only). To return <i>data</i> , echo it and capture with <code>\$ ()</code> .

12.8 Exit codes

```
mkdir newdir
echo $?          # 0 if it worked, non-zero if it did not

if ! mkdir labtest 2>/dev/null; then
    echo "Failed to create"
    exit 1
fi
echo "Created successfully"
exit 0
```

By convention 0 means success and anything else means failure. Give your own scripts meaningful exit codes — anything that runs them automatically will depend on it.

12.9 Debugging

Technique	Effect
<code>bash -x script.sh</code>	Trace every command as it runs, without editing the script.
<code>bash -n script.sh</code>	Check the syntax without running anything.
<code>set -x / set +x</code>	Trace just one section of the script.
<code>set -e</code>	Stop at the first command that fails.
<code>set -u</code>	Treat an unset variable as an error.
<code>shellcheck</code> <code>script.sh</code>	A linter that catches quoting and portability mistakes. Well worth installing.

12.10 Worked scripts

Five short scripts covering the techniques the laboratory tasks exercise: creating files in a loop, embedding system values, setting permissions programmatically, searching content, and moving files by pattern.

12.10.1 Creating records in a loop

Techniques: `mkdir -p`, a numeric `for` loop, command substitution for system values, appending with `>>`, and `$RANDOM` scaled to a range.

```
#!/bin/bash
# setup_students.sh -- create eight student record files
set -euo pipefail
```

```

DIR="$HOME/student_records"
OWNER="Ravindu"

mkdir -p "$DIR"
cd "$DIR"

for i in {1..8}; do
    file="student$i.txt"

    # system values, captured once per file
    {
        echo "Date:      $(date) "
        echo "Hostname: $(hostname) "
        echo "Username: $(whoami) "
        echo "Maintained by $OWNER"
    } > "$file"

    # a random mark between 40 and 100 inclusive
    mark=$(( RANDOM % 61 + 40 ))
    echo "Test score: $mark" >> "$file"

    echo "Created $file (score $mark)"
done

```

Scaling \$RANDOM

RANDOM % 61 gives 0–60; adding 40 shifts that to 40–100. The general form for a range is $((RANDOM \% (max - min + 1) + min))$.

12.10.2 Archiving and setting permissions

Techniques: creating a subdirectory, moving by wildcard, and applying a numeric mode.

```

#!/bin/bash
# archive_students.sh -- move the records into an archive directory
set -euo pipefail

DIR="$HOME/student_records"
ARCHIVE="$DIR/archive_records"

mkdir -p "$ARCHIVE"
cd "$DIR"

# move, not copy
mv /*.txt "$ARCHIVE/"

# owner rw, group rw, others r -> 664
chmod 664 "$ARCHIVE"/*.txt

ls -l "$ARCHIVE"

```

12.10.3 Searching content and counting matches

Techniques: `grep -l` to list matching *files*, redirecting into a report, and counting lines with `wc -l`.

```
#!/bin/bash
# filter_students.sh -- find records containing the digit 3
set -euo pipefail

DIR="$HOME/student_records/archive_records"
REPORT="$DIR/matched_students.txt"

cd "$DIR"

# -l prints the names of files that match, not the matching lines
grep -l "3" ./*.txt >> "$REPORT"

count=$(grep -l "3" ./*.txt | wc -l)
echo "Found $count matching files"
```

12.10.4 Generating a set of files from a list

Techniques: iterating over an explicit list, and building both the filename and the content from the loop variable.

```
#!/bin/bash
# setup_invoices.sh -- one invoice file per quarter
set -euo pipefail

DIR="$HOME/invoices_2025"
mkdir -p "$DIR"
cd "$DIR"

for q in Q1 Q2 Q3 Q4; do
    echo "This is the $q invoice summary" > "invoice_$q.txt"
done

ls -l
```

12.10.5 Selecting and moving by pattern

Techniques: matching alternatives with a bracket wildcard, and deleting selectively.

```
#!/bin/bash
# process_invoices.sh -- sort the quarterly invoices
set -euo pipefail

cd "$HOME/invoices_2025"

# display the odd quarters
echo "--- Q1 and Q3 ---"
cat invoice_Q[13].txt

# move the even quarters into a closed folder
mkdir -p closed
mv invoice_Q[24].txt closed/

# remove anything starting with invoice_Q1
rm -f invoice_Q1*

ls -R
```

Tip

`invoice_Q[13].txt` matches `invoice_Q1.txt` and `invoice_Q3.txt` but nothing else — more precise than two separate commands and less error-prone than `invoice_Q*.txt`.

12.11 Writing scripts that keep working

Habit	Why
<code>set -euo pipefail</code> at the top	A script that stops at the first error is far easier to debug than one that carries on with wrong data.
Quote every expansion	<code>"\$file"</code> , not <code>\$file</code> . Filenames with spaces are the most common cause of scripts that work in testing and fail in use.
Use absolute paths	A script's working directory at run time is not under your control, especially under <code>cron</code> .
Check before destroying	<code>[-d "\$dir"] exit 1</code> before an <code>rm -r</code> .
Comment the <i>why</i>	The commands say what happens; only you know why that was the right choice.
Test on a copy	Especially anything involving <code>rm</code> or <code>mv</code> .

Going further

Install `shellcheck` and run it over everything you write — it catches unquoted variables, useless uses of `cat` and subtle portability problems, and it explains each finding. From there: `trap` for cleaning up temporary files when a script is interrupted, `getopts` for parsing options properly, and `mktemp` for safe temporary files. And know the limit: once a script passes a few hundred lines or needs real data structures, rewrite it in Python.

PART IV

Development and Productivity Tools

CHAPTER 13

Git

Git records snapshots of a project over time. Nearly all of it follows from one picture — the three places a change can be — and this chapter is built around that picture.

13.1 What Git is for

Git is a *version control system*. It takes snapshots of a project so that you can:

- return to any previous state;
- see exactly what changed, when, and who changed it;
- work on a feature without breaking the version that works;
- collaborate without overwriting each other's work.

Term	Meaning
Repository (repo)	A project folder that Git is tracking.
Commit	A saved snapshot of the project at a point in time.
Branch	An independent line of work, so you can experiment safely.
Remote	A copy of the repository hosted elsewhere.

Git is not GitHub

Git is the tool, and it works entirely on your own machine with no network connection. GitHub is one website that hosts Git repositories — GitLab, Bitbucket and a server of your own do the same job. Everything in Sections 13.4 to 13.6 happens locally.

13.2 Before you start

```
git --version
```

This confirms Git is installed and shows the version, since a few commands differ slightly between them. Expect something like `git version 2.43.0`. If you get `command not found`, install Git first — `sudo apt install git` on Ubuntu, or from <https://git-scm.com/downloads> on Windows and macOS.

13.2.1 Identity

Every commit is permanently stamped with a name and an email address.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"

git config --global --list      # check what is set
git config --global core.editor vim
```

Note

Check with `git config --global user.name` before overwriting: if Git was set up on this machine before, you probably do not need to touch it. Omitting `--global` sets the value for the current repository only, which is how you use a different identity for one project without disturbing the rest.

13.3 The three trees

This is the model everything else rests on.

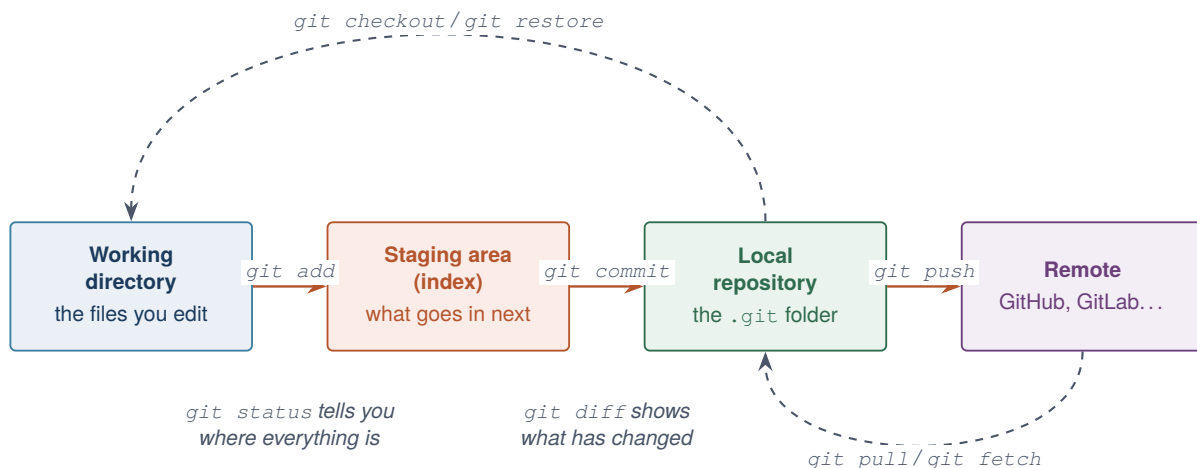


Figure 13.1. A change moves left to right. Nothing is permanent until it reaches the local repository, and nothing is shared until it reaches the remote.

Staging is a shopping basket

`git add` does not save anything — it selects which changes will go into the next snapshot. That is what lets you make five edits and commit three of them as one logical change, leaving the other two for a separate commit.

13.4 Your first repository

```
mkdir git-tutorial
cd git-tutorial
git init
```

`git init` turns an ordinary folder into a repository by creating a hidden `.git` directory — that is where the entire history will live from now on.

```
Initialized empty Git repository in /home/student/git-tutorial/.git/
```

```
# a per-repository identity, if you need one
git config user.name "Practice Account"
git config user.email "practice@example.com"

# create a file
echo "# My Practice Project" > README.md

git status
```

`git status` is the command you will run most. It tells you the state of every file and, usually, what to do next.

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  README.md
```

```
git add README.md          # stage it
git status                 # now shown under "Changes to be committed"

git commit -m "Add README" # take the snapshot
git log                   # see the history
```

The commit is permanent, timestamped and attributed to you.

13.5 The everyday cycle

One loop, repeated indefinitely: **edit** → **stage** → **commit**.

```
# 1. edit a file in any editor, then look at what changed
git diff                  # unstaged changes, line by line
git diff --staged         # what is staged, ready to commit

# 2. stage and commit
git add README.md
git commit -m "Describe project purpose"

# 3. stage everything at once
git add .
git commit -m "Add app.py and gitignore"

git log --oneline         # compact history, one line per commit
```

Command	Effect
<code>git add file</code>	Stage one file.
<code>git add .</code>	Stage every change in this directory and below.
<code>git add -A</code>	Stage every change in the whole repository.
<code>git add -p file</code>	Step through the changes and stage them in chunks.
<code>git commit -m "msg"</code>	Commit the staged changes with a message.
<code>git commit -am "msg"</code>	Stage all <i>tracked</i> files and commit in one step. Ignores new files.
<code>git commit --amend</code>	Rewrite the previous commit — to fix its message or add a forgotten file.
<code>git log --oneline</code>	Compact history.
<code>git log --graph</code>	History with the branch structure drawn.
<code>--oneline --all</code>	
<code>git show HEAD</code>	The full contents of the most recent commit.

Caution

Never `--amend` a commit that has already been pushed and that someone else may have pulled. It replaces the commit with a different one, and their history will no longer match yours.

13.5.1 .gitignore

Tells Git which files never to track — build output, logs, and anything secret.

```
cat > .gitignore <<'EOF'
*.log
__pycache__/
.env
node_modules/
*.aux
*.out
EOF

git add .gitignore
git commit -m "Add gitignore"
```

Caution

`.gitignore` only affects files Git is not *already* tracking. If a password has been committed once, adding it to `.gitignore` does nothing — it stays in the history for anyone who clones the repository. Rotate the credential; do not just delete the file.

13.5.2 Writing commit messages

Good	Poor
Add EMI calculator to Calc sheet	update
Fix off-by-one in student loop	fixed bug
Remove hard-coded discount rate	asdf

Write the subject line in the imperative — “Add”, not “Added” — keep it under about 50 characters, and if more explanation is needed, leave a blank line and write a paragraph beneath.

13.6 Branching and merging

A branch is an independent line of work. Commits made on it are invisible to the main line until you merge.

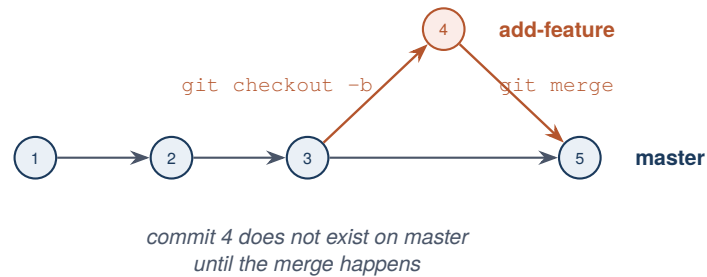


Figure 13.2. Work happens on `add-feature` in isolation; `git merge` brings it back.

```
git branch                # which branch am I on?
git checkout -b add-feature # create a branch and switch to it
                           # (git switch -c does the same, newer syntax)

# make a change on the branch
echo "print('new feature')" >> app.py
git add app.py
git commit -m "Add new feature to app.py"

# prove the isolation
git checkout master
cat app.py                # the new line is not here

# bring the work in
git merge add-feature
cat app.py                # now it is

git branch -d add-feature  # tidy up; the commits stay in history
```

master or main

Depending on your Git version and configuration, the default branch is called either `master` or `main`. Both are equally valid — it is only a name. These notes use `master`; substitute `main` if that is what `git branch` shows.

Command	Effect
<code>git branch</code>	List branches; the current one is marked.
<code>git branch name</code>	Create a branch without switching to it.
<code>git checkout name</code>	Switch to an existing branch.
<code>git checkout -b name</code>	Create and switch in one step.
<code>git switch name</code>	Newer, clearer equivalent of <code>checkout</code> .
<code>git merge name</code>	Merge that branch into the current one.
<code>git branch -d name</code>	Delete a merged branch.
<code>git branch -D name</code>	Delete it even if unmerged. Destroys the work.

13.6.1 Merge conflicts

A conflict happens when two branches changed the same lines. Git cannot decide which version is right, so it marks the disagreement in the file and stops:

```
<<<<<< HEAD
your version
=====
their version
>>>>>> branch-name
```

To resolve it: open the file, edit it so it contains the correct final content, delete all three marker lines, then:

```
git add <file>
git commit
```

Marker	Marks
<code><<<<<< HEAD</code>	The start of your version — what is on the branch you are merging <i>into</i> .
<code>=====</code>	The dividing line.
<code>>>>>>> name</code>	The end of their version — what is on the branch being merged <i>in</i> .

Tip

`git merge --abort` backs out of a conflicted merge entirely and returns you to the state before you started. Nothing is lost. Use it whenever a merge turns out to be more than you expected.

13.7 Working with a remote

13.7.1 Connecting to GitHub

Create an empty repository on GitHub first: click the + at the top right, choose **New repository**, name it `git-tutorial`, and leave **Add a README** unticked — you already have one locally. Then:

```
git remote add origin https://github.com/<your-username>/git-tutorial.git
git remote -v                # confirm where origin points
```

```
git push -u origin master      # first push: -u remembers the pairing
git push                      # every push after that
```

`origin` is just a conventional name for the main remote — `git remote add` is what tells Git where `push` and `pull` should send and fetch data.

13.7.2 Cloning and updating

Command	Effect
<code>git clone <url></code>	Copy an existing remote repository, with its full history, and set up <code>origin</code> automatically.
<code>git fetch</code>	Download new commits from the remote without changing your working files.
<code>git pull</code>	<code>fetch</code> followed by <code>merge</code> — download and integrate in one step.
<code>git push</code>	Send your commits to the remote.
<code>git push -u origin main</code>	Push and set the upstream branch.

Tip

`git pull` before you start work, and again before you push. Most conflicts come from working for a long time against a copy that has fallen behind.

Authentication

GitHub no longer accepts account passwords over HTTPS. Use a *personal access token* in place of the password, or set up SSH keys — Chapter 15 covers key generation, and the same key type works with GitHub. If two-factor authentication or account recovery blocks you during a lab session, creating a fresh practice account with a new email is the quickest way to keep moving.

13.8 Undoing things

Command	Undoes
<code>git restore file</code>	Discard uncommitted changes to a file.
<code>git restore --staged file</code>	Unstage a file, keeping the edit.
<code>git checkout -- file</code>	The older form of <code>git restore</code> .
<code>git commit --amend</code>	Change the last commit.
<code>git reset --soft HEAD~1</code>	Undo the last commit, keeping the changes staged.
<code>git reset --mixed HEAD~1</code>	Undo it, keeping the changes but unstaging them. The default.
<code>git reset --hard HEAD~1</code>	Undo it and throw the changes away.
<code>git revert <commit></code>	Create a <i>new</i> commit that reverses an old one. The safe option for shared history.
<code>git stash</code>	Put uncommitted work aside temporarily.
<code>git stash pop</code>	Bring it back.

Command	Undoes
<code>git reflog</code>	A log of everywhere HEAD has been — how you recover from a bad reset.

Caution

`git reset --hard` discards uncommitted work with no way back. `git stash` first if there is any doubt. On history that has already been pushed, use `git revert` rather than `reset` — it adds a commit instead of rewriting the past.

13.9 Command summary

Stage	Command	Purpose
Set up	<code>git init</code>	Start tracking a folder
	<code>git clone <url></code>	Copy an existing repository
	<code>git config</code>	Set your identity
Inspect	<code>git status</code>	What state is everything in
	<code>git diff</code>	What changed
	<code>git log</code>	What has been committed
Record	<code>git add</code>	Stage changes
	<code>git commit</code>	Save a snapshot
Branch	<code>git branch</code>	List or create branches
	<code>git checkout -b</code>	Create and switch
	<code>git merge</code>	Combine branches
Share	<code>git remote add</code>	Register a remote
	<code>git push</code>	Send commits up
	<code>git pull</code>	Bring commits down
Undo	<code>git restore</code>	Discard local changes
	<code>git revert</code>	Reverse a commit safely
	<code>git reflog</code>	Find something you lost

Going further

Practise deliberately creating a merge conflict — edit the same line of a file on two branches, merge them, and resolve it by hand. It is far less alarming once you have done it on purpose. After that: `git rebase` for a linear history, tags for marking releases, and pull requests as the collaboration workflow on GitHub. *Pro Git* by Chacon and Straub is free at <https://git-scm.com/book> and is the standard reference. Finally: commit often. Small commits are easy to understand, easy to revert and easy to merge.

CHAPTER 14

LaTeX

LaTeX is a document preparation system in which you describe a document's structure and let the software decide its appearance. This chapter covers the compilation model, document anatomy, and the constructions a technical report needs: sections, mathematics, tables, figures, cross-references and a bibliography.

14.1 What LaTeX is

In a word processor you make a heading look like a heading. In LaTeX you declare that something *is* a heading, and a set of typesetting rules decides how it should look. The consequences follow from that one difference: numbering, the table of contents, cross-references and the bibliography are all generated, because the document knows what its own parts are.

Name	What it is
T _E X	The typesetting engine, written by Donald Knuth. Powerful and extremely low-level.
L _A T _E X	A large collection of macros on top of T _E X, written by Leslie Lamport, that provides the document-level commands people actually use.
pdfL _A T _E X	The program that reads <code>.tex</code> source and produces a PDF.
Package	An add-on providing extra commands — <code>graphicx</code> for images, <code>amsmath</code> for mathematics.
Class	Defines the overall kind of document: <code>article</code> , <code>report</code> , <code>book</code> , <code>beamer</code> .

Note

LaTeX is worth learning for three situations in particular: documents with substantial mathematics, documents long enough that numbering and cross-referencing by hand becomes unreliable, and documents that must be regenerated reproducibly. Because the source is plain text, it also works perfectly with Git (Chapter 13).

14.2 The compilation cycle

You write source; a compiler produces the PDF. This is the part that surprises people arriving from a word processor.

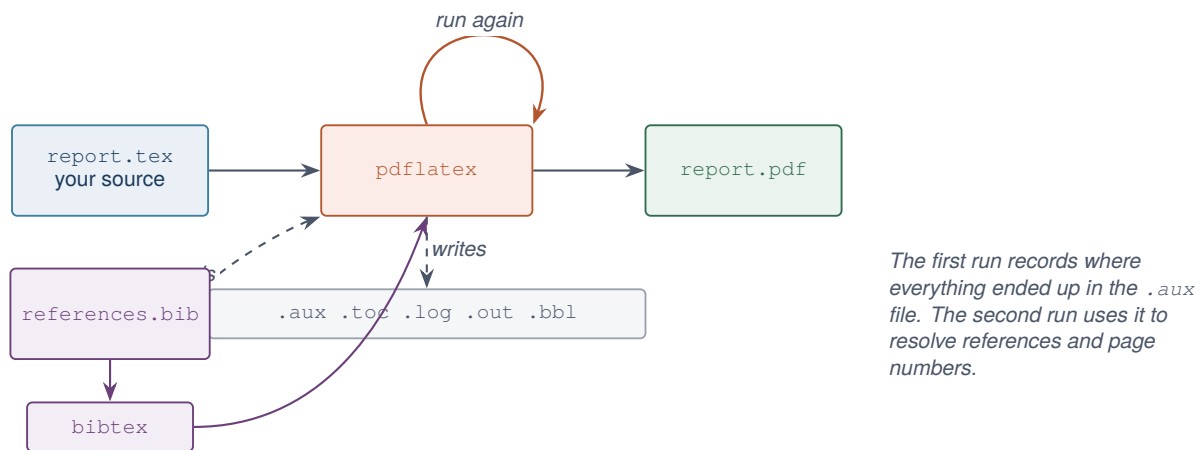


Figure 14.1. Cross-references need two passes, and a bibliography needs three. `latexmk` works out how many are required and runs them for you.

Command	Effect
<code>pdflatex report.tex</code>	One compilation pass.
<code>pdflatex report.tex</code> <code>pdflatex report.tex</code>	Two passes — needed whenever <code>\ref</code> , a table of contents or page numbers are involved.
<code>bibtex report</code>	Process the bibliography. Run between the first and second <code>pdflatex</code> .
<code>latexmk -pdf report.tex</code>	Works out how many passes are needed and runs them. Use this.
<code>latexmk -c</code>	Delete the auxiliary files, keeping the PDF.

References showing as ??

This almost always means you have compiled only once. The first pass writes the label positions to the `.aux` file; the second reads them back. Run it again — or use `latexmk`, which never gets this wrong.

14.3 Document anatomy

```

\documentclass[12pt,a4paper]{article}    % 1. class and options

\usepackage[utf8]{inputenc}              % 2. preamble: packages
\usepackage{graphicx}
\usepackage{amsmath}
\usepackage{hyperref}

\title{Application Laboratory Report}    % 3. title block metadata
\author{Your Name}
\date{\today}

\begin{document}                          % 4. the document body starts

\maketitle
\tableofcontents
\newpage

```

```
\section{Introduction}
Your text goes here.

\end{document} % 5. and ends
```

Region	Contains
<code>\documentclass</code>	Exactly one, on the first line. Options in square brackets set the base font size, paper size and layout.
Preamble	Everything between the class and <code>\begin{document}</code> : packages, custom commands, title metadata. Nothing here prints.
Body	Between <code>\begin{document}</code> and <code>\end{document}</code> . Everything that appears in the PDF.

14.3.1 Choosing a class

<code>article</code>	Papers and short reports. No chapters.
<code>report</code>	Longer reports. Adds <code>\chapter</code> .
<code>book</code>	Books. Chapters, plus front, main and back matter.
<code>beamer</code>	Presentation slides.

14.3.2 Useful packages

Package	Provides
<code>graphicx</code>	<code>\includegraphics</code> for images.
<code>amsmath</code>	Serious mathematics: <code>align</code> , <code>cases</code> , <code>\text</code> inside formulas.
<code>amssymb</code>	Additional mathematical symbols.
<code>hyperref</code>	Clickable cross-references, citations and URLs. Load it last.
<code>geometry</code>	Page margins: <code>\usepackage[margin=2.5cm]{geometry}</code> .
<code>booktabs</code>	Properly spaced horizontal rules for tables.
<code>listings</code>	Source code with syntax highlighting.
<code>babel</code>	Language-specific hyphenation and names.
<code>natbib</code>	Author–year citations, <code>\citet</code> and <code>\citep</code> .

14.4 The title page

```
\title{Application Laboratory Report}
\author{Your Name \\\ Index Number}
\date{\today} % or \date{4 August 2026}, or \date{} for none

\begin{document}
\maketitle
```

The first three commands only *record* the information; `\maketitle`, inside the document body, is what puts it on the page.

14.5 Table of contents

```
\tableofcontents
\newpage           % start the main content on a fresh page

\listoffigures     % optional
\listoftables      % optional
```

The contents list is built from your sectioning commands, so it needs no maintenance — but it does need two compilation passes.

14.6 Structure

Command	Level	Available in
<code>\part{}</code>	-1	all
<code>\chapter{}</code>	0	report, book
<code>\section{}</code>	1	all
<code>\subsection{}</code>	2	all
<code>\subsubsection{}</code>	3	all
<code>\paragraph{}</code>	4	all

Adding `*` suppresses both the number and the contents entry: `\section*{Acknowledgements}`.

14.7 Text formatting

Command	Produces
<code>\textbf{text}</code>	bold
<code>\textit{text}</code>	<i>italic</i>
<code>\emph{text}</code>	<i>emphasis</i> — italic, but flips to upright inside italic text
<code>\texttt{text}</code>	monospaced
<code>\underline{text}</code>	<u>underlined</u>
<code>\textsc{text}</code>	SMALL CAPITALS
<code>\footnote{text}</code>	a numbered footnote

Paragraphs and spacing

A blank line starts a new paragraph. Two blank lines do exactly the same thing — $\text{\LaTeX}$ controls vertical spacing itself. Likewise, multiple spaces between words collapse into one. Resisting the urge to adjust spacing by hand is most of learning $\text{\LaTeX}$.

14.8 Lists

```
\begin{itemize}
\item First bulleted point
```

```

\item Second point
\begin{itemize}
\item A nested point
\end{itemize}
\end{itemize}

\begin{enumerate}
\item First numbered point
\item Second numbered point
\end{enumerate}

\begin{description}
\item[Term] Its definition.
\item[Another] Another definition.
\end{description}

```

14.9 Mathematics

14.9.1 Inline and displayed

Einstein's relation $E = mc^2$ appears in the flow of the sentence.

The quadratic formula is displayed on its own line:

```

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

```

A numbered equation that can be referred to later:

```

\begin{equation}
\label{eq:quadratic}
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{equation}

```

Which typesets as: Einstein's relation $E = mc^2$ inline, and

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

14.9.2 Common constructions

Source	Output	Source	Output
x^2	x^2	α	α
x_i	x_i	β	β
$\frac{a}{b}$	$\frac{a}{b}$	π	π
$\sqrt{x}$	$\sqrt{x}$	$\times$	$\times$
$\sum_{i=1}^n$	$\sum_{i=1}^n$	$\leq$	$\leq$
$\int_a^b$	$\int_a^b$	$\geq$	$\geq$
$\pm$	$\pm$	$\neq$	$\neq$
∞	∞	$\approx$	$\approx$

```
% several aligned equations, from amsmath
```

```
\begin{align}
f(x) &= (x+1)^2 \\
&= x^2 + 2x + 1
\end{align}
```

Caution

Subscripts and superscripts apply to one character unless you group them. x^{10} gives x^10 ; $x^{\{10\}}$ gives x^{10} .

14.10 Tables

```
\begin{table}[h]
\centering
\caption{Student marks by subject}
\label{tab:marks}
\begin{tabular}{|l|c|c|c|}
\hline
Name & Maths & English & Science \\
\hline
Student A & 78 & 85 & 90 \\
Student B & 65 & 70 & 68 \\
Student C & 90 & 88 & 92 \\
\hline
\end{tabular}
\end{table}
```

Spec	Column
l	Left aligned, width set by content.
c	Centred.
r	Right aligned.
p{3cm}	Fixed width, text wraps.
	A vertical rule between columns.

Inside the table, & separates cells, \\ ends a row, and \hline draws a horizontal rule.

Tip

The `booktabs` package gives `\toprule`, `\midrule` and `\bottomrule`, with no vertical rules at all. Every table in these notes uses it. The result reads better than a fully boxed grid, and it is the standard in published work.

table versus tabular

`tabular` draws the grid. `table` is the *float* that wraps it, providing the caption, the label and the ability to move to a convenient place on the page. You need both for a table you intend to refer to.

14.11 Figures

```
\usepackage{graphicx}           % in the preamble

\begin{figure}[h]
  \centering
  \includegraphics[width=0.7\textwidth]{diagram.png}
  \caption{System architecture}
  \label{fig:architecture}
\end{figure}
```

Option	Effect
<code>width=0.7\textwidth</code>	Scale to 70% of the text width. Prefer this to absolute sizes — it survives a change of page layout.
<code>scale=0.5</code>	Half the natural size.
<code>angle=90</code>	Rotate.

The placement specifier in square brackets is a *request*: `h` here, `t` top of page, `b` bottom, `p` its own page, `!` try harder. `H` (from the `float` package) forces the exact position.

Tip

Do not fight the float placement. $\text{\LaTeX}$ moves figures to avoid leaving large gaps, and because you refer to them by number rather than by “the figure below”, it does not matter where they land.

14.12 Cross-referencing

```
\section{Methodology}
\label{sec:method}
```

As shown in Table~\ref{tab:marks}, the results are consistent.
 Figure~\ref{fig:architecture} illustrates the design.
 The method is described in Section~\ref{sec:method} on
 page~\pageref{sec:method}.

Command	Inserts
<code>\label{key}</code>	An invisible marker. Place it immediately after the caption or the sectioning command.
<code>\ref{key}</code>	The number of whatever was labelled.
<code>\pageref{key}</code>	The page number it appears on.

Two conventions worth adopting

Use a non-breaking space — Table~\ref{tab:marks} — so the word and the number never end up on different lines. And prefix your keys by type: `sec:`, `fig:`, `tab:`, `eq:`. In a long document this is the difference between a label scheme you can use and one you have to look up.

14.13 Citations and bibliography

Create a separate file, `references.bib`:

```

book{shotts2019,
  author   = {William E. Shotts},
  title    = {The Linux Command Line},
  publisher = {No Starch Press},
  year     = {2019},
  edition  = {2nd}
}

article{lamport1978,
  author = {Leslie Lamport},
  title  = {Time, Clocks, and the Ordering of Events},
  journal = {Communications of the ACM},
  volume = {21},
  number = {7},
  pages  = {558--565},
  year   = {1978}
}

```

Then, in the document:

```

The shell is covered thoroughly by \cite{shotts2019}, and
\citet{lamport1978} introduced logical clocks.

\bibliographystyle{plain}      % or unsrt, alpha, abbrv, apalike
\bibliography{references}      % the .bib file, without the extension

```

Compile with: `pdflatex`, then `bibtex`, then `pdflatex` twice — or simply `latexmk -pdf`.

Entry type	For
@book	Books.
@article	Journal articles.
@inproceedings	Conference papers.
@misc	Web pages and anything that fits nowhere else.
@techreport	Technical reports.

Note

`\cite` produces a bracketed reference; `\citet` and `\citep` (from `natbib`) give textual and parenthetical author–year forms. Only works cited actually appear in the bibliography — add `\nocite{*}` to include everything in the `.bib` file.

14.14 Special characters

Ten characters have meaning to $\text{\LaTeX}$ and must be escaped to appear literally.

Want	Type	Want	Type
%	<code>\%</code>	&	<code>\&</code>
\$	<code>\\$</code>	#	<code>\#</code>
_	<code>_</code>	{	<code>\{</code>
}	<code>\}</code>	~	<code>\textasciitilde</code>
^	<code>\textasciicircum</code>	\	<code>\textbackslash</code>

Caution

% starts a comment: everything after it on that line is ignored. An unescaped % in a percentage is the most common way to make half a line silently vanish.

14.14.1 Quotation marks and dashes

Type	Get	Use for
<code>'single'</code>	<code>'single'</code>	single quotation
<code>"double"</code>	<code>"double"</code>	double quotation
<code>-</code>	<code>-</code>	hyphen: <i>well-known</i>
<code>--</code>	<code>-</code>	en dash: ranges, <i>pages 5–10</i>
<code>---</code>	<code>—</code>	em dash: an aside — like this one

14.15 Common errors

Message	Usual cause
Undefined control sequence	A misspelled command, or a package you forgot to load.
Missing \$ inserted	A mathematical symbol used outside maths mode — an underscore or a caret in ordinary text.
File not found	An image or package that is not where L ^A T _E X is looking.
Missing \begin{document}	Text placed in the preamble.
LaTeX Warning: Reference 'x' undefined	The label does not exist, or you have compiled only once.
Overfull \hbox	A line is slightly too wide — often a long unbreakable word or URL. Usually cosmetic.
Runaway argument	An unclosed brace.

Tip

Compile early and often. A L^AT_EX error report points at the line where the compiler noticed the problem, which may be well after the line that caused it. Compiling after every few paragraphs keeps the search short.

14.16 A complete report skeleton

Everything above, assembled into a document you could submit.

```
\documentclass[12pt,a4paper]{article}

\usepackage[utf8]{inputenc}
\usepackage[margin=2.5cm]{geometry}
\usepackage{graphicx}
\usepackage{amsmath}
\usepackage{booktabs}
```

```

\usepackage{hyperref}

\title{Application Laboratory Report}
\author{Your Name \& Index Number}
\date{\today}

\begin{document}

\maketitle
\tableofcontents
\newpage

\section{Introduction}
\subsection{Background}
Opening text. See Table~\ref{tab:results}.

\section{Methodology}
\subsection{Procedure}
\begin{enumerate}
  \item First step
  \item Second step
\end{enumerate}

The relationship is given by  $E = mc^2$ , and the roots follow from
\begin{equation}
  x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{equation}

\begin{table}[h]
  \centering
  \caption{Results}
  \label{tab:results}
  \begin{tabular}{lcc}
    \toprule
    Sample & Mass (g) & Yield (%) \\
    \midrule
    A & 12.4 & 87 \\
    B & 15.1 & 91 \\
    \bottomrule
  \end{tabular}
\end{table}

\begin{figure}[h]
  \centering
  \includegraphics[width=0.6\textwidth]{diagram.png}
  \caption{Experimental setup}
  \label{fig:setup}
\end{figure}

As Figure~\ref{fig:setup} shows, the apparatus is straightforward.
This approach follows \cite{shotts2019}.

\bibliographystyle{plain}
\bibliography{references}

\end{document}

```

Going further

Overleaf (<https://www.overleaf.com>) is a browser-based $\text{\LaTeX}$ editor that needs no installation and compiles as you type — the fastest way to start. For a local setup, install TeX Live and use VS Code with the LaTeX Workshop extension. The *Not So Short Introduction to $\text{\LaTeX}$* is the standard free tutorial, and <https://tex.stackexchange.com> answers nearly everything else. When you are comfortable, look at TikZ for drawing diagrams in the source (every diagram in these notes is TikZ), Beamer for presentations, and BibLaTeX with Biber as the modern replacement for BibTeX.

CHAPTER 15

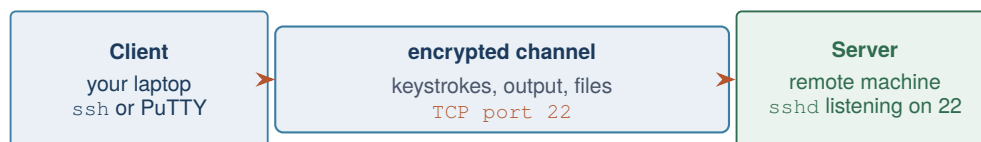
SSH — Secure Shell

SSH gives you a shell on a machine you are not sitting in front of, over an encrypted connection. This chapter covers the protocol, key-based authentication, file transfer, configuration and hardening, then the same ground from Windows using PuTTY.

15.1 What SSH is

SSH (Secure Shell) is a cryptographic network protocol for secure remote login and command execution. It replaces older protocols such as Telnet, rlogin and FTP, all of which transmit data — including passwords — in plain text where anyone on the network path can read them.

- Encrypts everything, passwords included.
- Provides secure file transfer through `scp` (Secure Copy Protocol) and `sftp` (SSH File Transfer Protocol).
- Enables secure remote administration, and can tunnel other protocols through the same encrypted channel.



anyone intercepting this traffic sees only ciphertext

Figure 15.1. The client opens an encrypted channel to the server’s SSH daemon. Everything — authentication, commands, output and file transfers — travels inside it.

15.2 Installing

15.2.1 The server

To *accept* connections, a machine needs the SSH server installed.

```
sudo apt update
sudo apt install openssh-server
```

15.2.2 The client

Most Linux systems have it already. Check with:

```
ssh -V
```

If it is missing:

```
sudo apt install openssh-client
```

15.2.3 Starting and managing the service

```
sudo systemctl start ssh      # start it now
sudo systemctl enable ssh     # start it automatically at boot
sudo systemctl status ssh     # is it running?
sudo systemctl restart ssh    # after changing the configuration
```

A healthy service reports something like:

```
ssh.service - OpenBSD Secure Shell server
Loaded: loaded (/lib/systemd/system/ssh.service; enabled)
Active: active (running) since Tue 2026-08-04 09:12:31 UTC
```

Note

`start` and `enable` are separate. `start` affects this boot; `enable` affects the next one. Run both.

15.3 Connecting

Syntax

```
ssh <username>@<hostname-or-ip>
```

```
ssh alice@192.168.1.100
ssh alice@server.example.com
ssh -p 2222 alice@192.168.1.100  # a non-default port
```

On the first connection you are asked to confirm the server's identity:

```
The authenticity of host '192.168.1.100' can't be established.
ED25519 key fingerprint is SHA256:abc123...
Are you sure you want to continue connecting (yes/no)?
```

Answering `yes` records the server's key in `~/.ssh/known_hosts`. From then on, SSH checks that the key matches — which is what prevents someone impersonating the server later. Then you are asked for the password.

15.3.1 Running a single remote command

Instead of opening a full interactive session, append the command:

```
ssh alice@192.168.1.100 "uptime"
ssh alice@192.168.1.100 "df -h"
ssh alice@192.168.1.100 "cat /etc/os-release"
```

The command runs on the remote machine, its output is printed locally, and the connection closes. This is what makes SSH scriptable.

15.3.2 Useful options

Option	Effect
<code>-p</code>	Specify the port: <code>ssh -p 2222 user@host</code> .
<code>-v</code>	Verbose output. Repeat as <code>-vvv</code> for more. The first thing to reach for when a connection fails.
<code>-X</code>	Enable X11 forwarding, so remote graphical applications display locally.
<code>-N</code>	Do not execute a remote command — used when setting up port forwarding.
<code>-i</code>	Use a specific private key file.
<code>-L</code>	Local port forwarding.

15.4 Key-based authentication

Passwords can be guessed, and typing one for every connection makes automation impossible. Key authentication replaces them with a pair of mathematically related files.

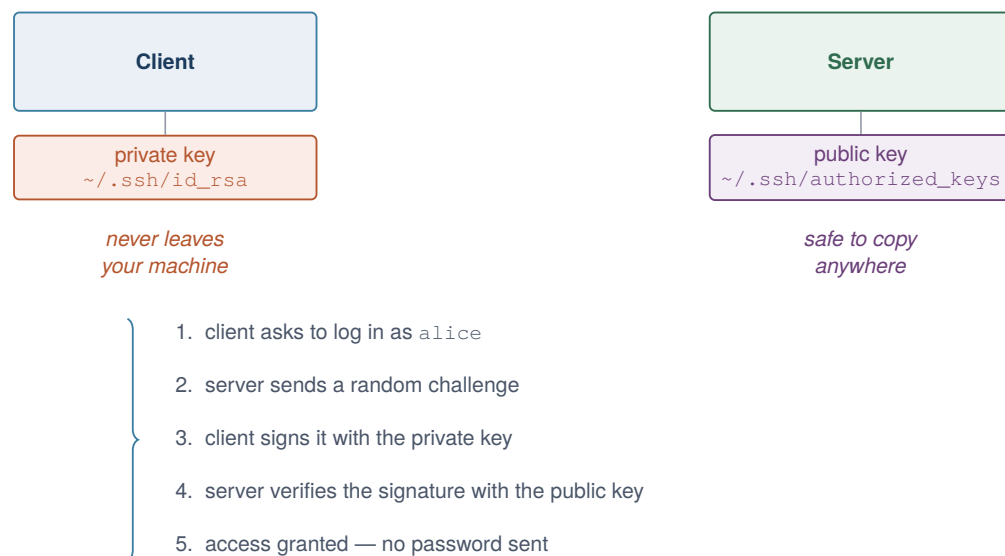


Figure 15.2. The private key is never transmitted. The server only ever sees a signature it can check against the public key it already holds.

15.4.1 Generating a key pair

```
ssh-keygen -t rsa -b 4096
# or, preferred on modern systems:
ssh-keygen -t ed25519
```

By default this creates two files:

File	Role
<code>~/.ssh/id_rsa</code>	The private key. Keep it secret. Never copy it anywhere, never commit it to Git.
<code>~/.ssh/id_rsa.pub</code>	The public key. Distribute it freely — it is designed to be published.

You are offered a *passphrase*. Setting one encrypts the private key on disk, so that a stolen laptop does not hand over your servers.

15.4.2 Installing the public key on the server

```
ssh-copy-id alice@192.168.1.100
```

That appends your public key to `~/.ssh/authorized_keys` on the remote machine, creating the directory and setting the permissions correctly. From then on you log in without a password.

Doing it by hand — when `ssh-copy-id` is unavailable, as on Windows:

```
mkdir -p ~/.ssh
nano ~/.ssh/authorized_keys      # paste the public key on its own line
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Permissions are enforced

SSH refuses to use keys that other users could read. `~/.ssh` must be `700` and `authorized_keys` and private keys must be `600`. If key authentication silently falls back to asking for a password, check these first — `ssh -v` will say so explicitly.

15.5 Transferring files

15.5.1 scp

Syntax

```
scp [options] <source> <destination>
Either side may be remote, written as user@host:/path.
```

```
# local -> remote
scp file.txt alice@192.168.1.100:/home/alice/

# remote -> local (the dot means "here")
scp alice@192.168.1.100:/home/alice/file.txt .

# a whole directory
scp -r project/ alice@192.168.1.100:/home/alice/

# through a non-default port (capital P, unlike ssh)
scp -P 2222 file.txt alice@192.168.1.100:/home/alice/
```

Note

`scp` uses `-P` for the port while `ssh` uses `-p`. It is an inconsistency in the tools, not a typo. For large or repeated transfers, `rsync -avz -e ssh` is better — it copies only the differences and can resume.

15.5.2 sftp

An interactive session, like FTP but encrypted.

```
sftp alice@192.168.1.100
```

<code>ls</code>	list the remote directory	<code>lls</code>	list the local one
<code>cd</code>	change remote directory	<code>lcd</code>	change local directory
<code>get</code>	download a file	<code>put</code>	upload a file
<code>pwd</code>	remote working directory	<code>bye</code>	disconnect

15.6 The client configuration file

`~/.ssh/config` saves you retyping connection details.

```
Host myserver
  HostName 192.168.1.100
  User alice
  Port 22
  IdentityFile ~/.ssh/id_rsa

Host lab
  HostName lab.university.edu
  User student
  Port 2222
```

Then simply:

```
ssh myserver
scp report.txt myserver:/home/alice/
```

15.7 Hardening the server

Edit `/etc/ssh/sshd_config` as root, then restart the service.

Setting	Effect
<code>Port 2222</code>	Move off port 22. Cuts automated scanning noise dramatically, though it is obscurity rather than security.
<code>PermitRootLogin no</code>	Stop anyone logging in directly as root. They must log in as themselves and use <code>sudo</code> , which is also auditable.
<code>PasswordAuthentication no</code>	Keys only. The single most effective change — but set up and test a key first.
<code>AllowUsers alice bob</code>	Only these accounts may log in at all.
<code>MaxAuthTries 3</code>	Drop the connection after three failures.

```
sudo nano /etc/ssh/sshd_config
sudo sshd -t # check the syntax BEFORE restarting
sudo systemctl restart ssh
```

Caution

Keep your existing session open while you test the new configuration from a second terminal. A mistake in `sshd_config` that only bites on the next connection will otherwise lock you out of a remote machine permanently. `sudo sshd -t` validates the file without applying it.

15.8 Troubleshooting

Symptom	Check
Connection refused	Nothing is listening. Is the service running, and is the address right? <code>sudo systemctl status ssh</code> , then <code>sudo ss -tln grep :22</code> . Expect a line showing <code>LISTEN ... 0.0.0.0:22</code> . If the service is stopped: <code>sudo systemctl start ssh</code> and <code>sudo systemctl enable ssh</code> .
Connection timed out	Something is silently dropping the traffic. Is the machine running? Check the firewall: <code>sudo ufw status</code> should show <code>22/tcp ALLOW Anywhere</code> . If not: <code>sudo ufw allow 22</code> then <code>sudo ufw reload</code> . On a cloud VM, check the security group as well.
Permission denied (publickey)	Key authentication failed and passwords are disabled. Is the public key in the right user's <code>authorized_keys</code> ? Are the permissions 700 and 600? Run <code>ssh -v</code> to see which keys were offered.
Permission denied (password)	Wrong username or password. Note that the username on the server is often not the one you use locally.
Host key verification failed	The server's key has changed since you last connected. If the machine was legitimately rebuilt, remove the stale entry: <code>ssh-keygen -R 192.168.1.100</code> . If it was not, stop and investigate.
Host key alert on first connect	Normal. Answer <code>yes</code> — provided you are connecting to what you think you are.

Tip

`ssh -vvv user@host` prints every step of the negotiation: which keys were tried, which authentication methods the server offered, and exactly where it gave up. It answers most SSH questions faster than any amount of guessing.

15.9 SSH from Windows with PuTTY

PuTTY is a free, open-source SSH client for Windows providing terminal emulation for connecting securely to Linux systems.

Note

Windows 10 and later include OpenSSH: `ssh user@host` works in PowerShell exactly as it does on Linux. PuTTY remains useful for its saved sessions, its key manager and on older systems.

15.9.1 Installing

- Go to <https://www.putty.org>.
- Download the Windows installer, `putty-64bit-<version>.msi`.
- Install with the default settings.
- Launch PuTTY from the Start menu.

15.9.2 Connecting with a password

1. Open PuTTY.
2. In **Host Name (or IP address)**, enter the IP or domain of the Linux machine.
3. Set **Port** to 22.
4. Ensure **Connection type** is **SSH**.
5. Optionally, type a name under **Saved Sessions** — `MyLinuxVM` — and click **Save**.
6. Click **Open**.
7. If a security alert appears on the first connection, click **Yes** to trust the host.
8. Enter your username, press **Enter**, then the password. Nothing is echoed as you type it.

15.9.3 Keys with PuTTYgen

PuTTY uses its own private key format, `.ppk`.

Step 1 — generate

Open PuTTYgen from the Start menu. Choose **RSA**, click **Generate**, and move the mouse around the blank area to supply randomness. Then save the **private key** as a `.ppk` file, and copy the **public key** text from the box at the top of the window.

Step 2 — install the public key

Get it onto the server by some other means — an existing password session, SFTP via FileZilla, or a cloud console — and append it to `authorized_keys`:

```
mkdir -p ~/.ssh
nano ~/.ssh/authorized_keys      # paste the public key, then save
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Step 3 — point PuTTY at the key

In PuTTY go to **Connection** → **SSH** → **Auth**, click **Browse** and select the `.ppk` file. Return to **Session**, select your saved session and click **Save** so the setting is remembered. Click **Open** to connect with the key.

15.9.4 Transferring files from Windows

PuTTY has no graphical file transfer. Two options:

PSCP — Download `pscp.exe` from the PuTTY site and place it in `C:\Windows\System32` so it is on the path.

```
:: Windows to Linux
pscp C:\Users\user\Documents\hello.txt username@192.168.1.100:/home/username/

:: Linux to Windows
pscp username@192.168.1.100:/home/username/report.txt C:\Users\user\Downloads\
```

FileZilla — Install FileZilla, choose the **SFTP** protocol, and enter `sftp://<ip>` as the host, your username, your password, and port 22. This gives a two-panel drag-and-drop interface.

15.9.5 Useful PuTTY settings

Setting	Where and why
Keep session alive	Connection tab; set Seconds between keepalives to 30 to stop idle connections being dropped.
Auto-login username	Connection → Data ; set Auto-login username to <code>ubuntu</code> or whichever account you use.

15.10 Connecting to a virtual machine

Whether the VM is local (VirtualBox, VMware) or in the cloud, the requirements are the same:

You need	Typical value
IP address or hostname	<code>192.168.1.101</code> , or a public cloud IP. Find it from inside the VM with <code>ip a</code> .
Port	22 unless changed.
Username	<code>ubuntu</code> , <code>ec2-user</code> , <code>centos</code> , <code>admin</code> , <code>root</code> , or one you created. Cloud images each have their own convention.
Authentication	The user's password, or a private key — a <code>.ppk</code> file if you are using PuTTY.

Once connected, a few commands confirm where you are:

```
whoami           # which account am I?
hostname         # which machine?
uname -a         # kernel and architecture
uptime           # how long has it been running
df -h           # disk usage
ls -al ~         # what is in this account's home directory
sudo apt update  # can I administer it?
```

Going further

Three things extend SSH considerably. *Port forwarding* (`-L`, `-R`) tunnels other services through the encrypted channel — reaching a database that only listens on the server’s loopback interface, for instance. `ssh-agent` holds your decrypted private key in memory so a passphrase-protected key still only needs the passphrase once per session. `tmux` or `screen` keeps a remote session alive across disconnections, so a long-running job survives your laptop closing. Also worth installing on any internet-facing server: `fail2ban`, which watches the authentication log and blocks addresses that keep guessing.

CHAPTER 16

Freeplane and Mind Mapping

A mind map is a diagram of ideas arranged around a centre. Freeplane is a free, open-source tool for building them. This chapter covers the technique, the keyboard-driven workflow that makes it fast, and how to structure a map that has to hold contradictory information.

16.1 What mind mapping is for

A mind map places one central idea in the middle and branches outwards into progressively more specific sub-ideas. It suits three situations that linear notes handle badly:

- **The structure is not known in advance.** You can add a branch anywhere without rewriting what came before.
- **Relationships cross the hierarchy.** A clue may bear on three different suspects; a requirement may affect four components. A *cross-link* draws that connection directly.
- **The whole must be visible at once.** Collapsing branches lets a large body of material fit on one screen.

Note

Freeplane began as a fork of FreeMind and has since diverged considerably — richer formatting, scripting, conditional styles and better export. Maps are saved in the `.mm` format, which both tools can read. The instructions here are for Freeplane 1.x, verified against the current stable 1.13 series, and use only menu items and shortcuts that have been stable across the 1.x line.

16.2 Anatomy of a map

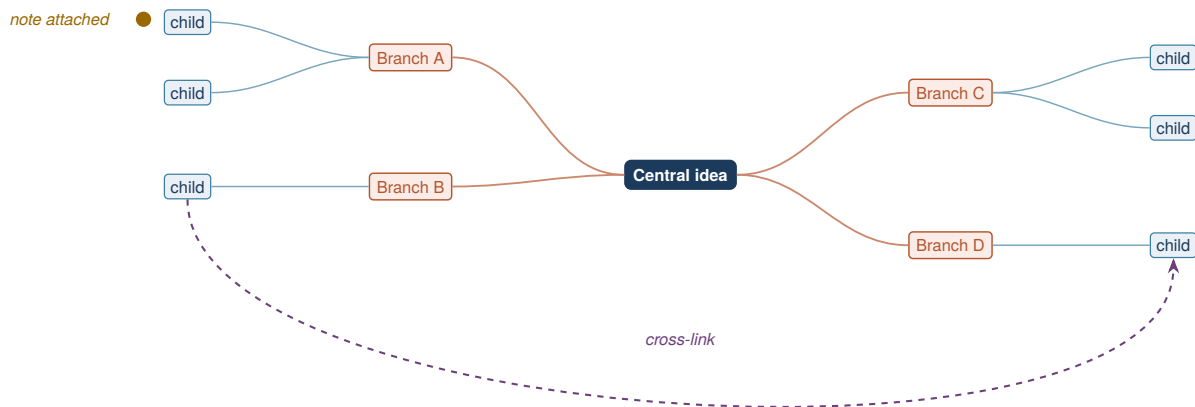


Figure 16.1. Nodes form a tree radiating from the centre. A cross-link connects two nodes that are not related in the tree — which is what the tree alone cannot express.

Element	What it is
Central node	The root. Every map has exactly one, and it names the subject.
Child	A node one level deeper — a subdivision of its parent.
Sibling	A node at the same level, sharing a parent.
Edge	The line joining a parent to a child.
Cross-link	An arrow between two nodes anywhere in the map, independent of the tree.
Note	Longer text attached to a node but displayed separately, so the node label stays short.
Icon	A small symbol on a node, used to mark status or category.

16.3 The core workflow

Freeplane is built around two keys. Learning them is most of learning the tool.

Key	Creates
Insert	A child of the selected node — one level deeper.
Enter	A sibling of the selected node — the same level.

Tip

Enter does double duty: it creates a sibling when a node is *selected*, and it confirms the text when you are *editing* a node. Which one happens depends on whether you are in edit mode — the same modal distinction as Vim in Chapter 9.

16.3.1 Building a map, step by step

1. **New map.** **File** → **New**. A new map opens with a single central node called **New Mindmap**.

2. **Rename the centre.** Double-click it, delete the existing text, type `My First Map`, and press **Enter** to confirm.
3. **Add children.** Single-click the central node to select it — one click, not two. Press **Insert**; a new child node appears ready for text. Type `Idea 1` and press **Enter**. Click the central node again and repeat for `Idea 2` and `Idea 3`. You now have three children attached to the centre.
4. **Add a sibling.** Click `Idea 1` to select it and press **Enter**. This creates a node at the *same* level — a sibling, not a child. Type `Idea 1b` and confirm.
5. **Add a grandchild.** Click `Idea 2`, press **Insert**, type `Sub-point A` and confirm.
6. **Move a node by dragging.** Click and hold `Idea 1b`, drag it on top of `Idea 3` and release. `Idea 1b` is now a child of `Idea 3` rather than a sibling of `Idea 1`.
7. **Colour a node and its branch.** Right-click `Idea 2` and go to **Format** → **Node Color** to change the text colour — or use the paint-bucket and **A** icons on the toolbar. Right-click it again and choose **Format** → **Edge Color** to colour the branch line leading to it.
8. **Add an icon.** Right-click `Idea 3`, go to **Icons** in the context menu (or the **Icons** menu in the menu bar) and choose one — a star or a flag. It appears beside the node text.
9. **Attach a note.** Select `Sub-point A`. Open the note editor with **View** → **Note Panel**, or right-click the node and choose **Notes**; the editing area appears at the bottom or side of the screen. Click into it and type the longer text. Click back on the node and a small note icon now marks it.
10. **Create a cross-link.** Click `Idea 1`, then hold **Ctrl** and click `Idea 3` so both are selected — the second one clicked becomes the arrow's target. Right-click either and choose **Connect**. A curved arrow now joins two nodes that are not connected in the tree.
11. **Collapse and expand.** Click the small circle at the edge of `Idea 2`, next to its child. The branch collapses, hiding its children; click again to expand it.
12. **Save.** **File** → **Save As**, and save as `firstmap.mm`.

16.3.2 Three ways to make a cross-link

Versions differ slightly in where the command lives, so it is worth knowing all three:

- Select both nodes with **Ctrl**-click, then right-click → **Connect**.
- Select both nodes and press **Ctrl+L**.
- Hold **Ctrl+Shift** and drag one node directly onto the other. This works in every version.

If **Connect** is not in the context menu, look under the **Insert** menu in the menu bar instead.

16.4 Keyboard reference

Key	Action
Insert	New child node
Enter	New sibling node (or confirm text while editing)
Shift+Enter	New sibling <i>before</i> the current node
F2	Edit the selected node
Esc	Cancel editing

Key	Action
Delete	Delete the node and everything below it
Arrow keys	Move between nodes
Ctrl+click	Add a node to the selection
Ctrl+L	Create a cross-link between selected nodes
Ctrl+Shift+drag	Create a cross-link by dragging
Space	Collapse or expand a branch
Ctrl+Z	Undo
Ctrl+S	Save
Ctrl+F	Find

Tip

Build with the keyboard, format with the mouse. Typing **Insert**, text, **Enter**, text, **Enter** keeps pace with your thinking; reaching for the mouse between every node does not. Do all the colouring and icon work at the end, once the structure has settled.

16.5 Formatting for meaning

Colour and icons are only useful if they mean something consistently.

Device	Where	Use it to encode
Node colour	Format → Node Color	The category a node belongs to.
Edge colour	Format → Edge Color	Which main branch a subtree descends from.
Icons	Icons menu or right-click	Status: confirmed, uncertain, ruled out, priority.
Node style	Format → Node Shape	Kind of thing: fact, question, conclusion.
Notes	View → Note Panel	Detail that would make the node label unreadable.

Keep node labels short

A node should be a label, not a sentence. Put the full text in a *note* attached to the node. The map then stays readable at a glance, and the detail is one click away whenever you need it.

16.6 Structuring a map for reasoning

A mind map is at its most useful when the material is incomplete or contradictory — the point at which a flat list of facts stops helping. The structure below generalises to any investigation, requirements analysis or literature review.

Element	Purpose
A central node naming the question	Everything in the map must relate to it. If something does not, it belongs in a different map.
Main branches for each <i>kind</i> of thing	Candidates, evidence, locations, timeline. Categories, not conclusions.
One colour per candidate	So that a glance shows which parts of the map bear on which possibility.
Full evidence text in <i>notes</i>	The node carries a short label; the note carries the wording. This is what keeps a large map legible.
Status icons on every piece of evidence	A tick for confirmed, a question mark for ambiguous, a cross for ruled out. Ambiguity gets recorded rather than resolved prematurely.
Cross-links from evidence to candidates	One piece of evidence may link to several candidates. These links are the reasoning; the tree is only the filing system.
A conclusion node	Stating the answer, cross-linked to the specific evidence that supports it — so the justification is visible, not remembered.

Contradictory evidence is the point

Real material contains items that cut both ways, and items that look decisive but are not. Mark those with a question-mark icon and cross-link them to *every* candidate they touch, rather than filing each one under a single heading. Forcing yourself to weigh conflicting evidence is precisely what the cross-link structure is for — a flat list quietly hides the conflict.

16.7 Saving and exporting

Freeplane's native format is `.mm` — an XML file, which means it diffs and merges sensibly under Git.

Format	Use for
<code>.mm</code>	Working format. Keep this one.
PNG / JPEG	Dropping the map into a report or slide.
PDF / SVG	Printing, or scaling without loss of quality.
HTML	Sharing with someone who does not have Freeplane.
OpenDocument	Converting the map into a linear outline in Writer.

Export through **File** → **Export map**.

Going further

Two features repay exploration once the basics are automatic. *Conditional styles* apply formatting automatically based on a node's content or icons — so every node tagged as a risk turns red without your touching it. *Filtering* hides everything that does not match a condition, which turns a large map into a series of focused views. Freeplane also scripts in Groovy for bulk operations. More broadly: mind maps are excellent for generating and organising ideas, and poor for anything requiring precise sequence or strict logic — for those, reach for a flowchart, an outline or a proper document instead.

PART V

Appendices

Command and Shortcut Reference

Everything in one place, for looking up rather than reading. Each entry points back to the chapter that explains it.

A.1 Navigation and inspection

Command	Purpose
<code>pwd</code>	Print the working directory
<code>cd <dir></code>	Change directory
<code>cd ..</code>	Up one level
<code>cd ~</code> or <code>cd</code>	Home
<code>cd -</code>	Previous directory
<code>pushd <dir></code>	Push onto the directory stack and go there
<code>popd</code>	Pop and return
<code>dirs -v</code>	Show the directory stack
<code>ls</code>	List
<code>ls -l</code>	Long listing
<code>ls -a</code>	Include hidden files
<code>ls -lh</code>	Human-readable sizes
<code>ls -R</code>	Recursive
<code>ls -lt</code>	Sort by modification time
<code>file <name></code>	Identify a file's actual type
<code>stat <name></code>	Full metadata, including numeric permissions
<code>which <cmd></code>	Path of the program that would run
<code>type <cmd></code>	Program, builtin, alias or function?

A.2 Files and directories

Command	Purpose
<code>mkdir <dir></code>	Create a directory

Command	Purpose
<code>mkdir -p a/b/c</code>	Create a chain, making parents as needed
<code>mkdir -p x/{a,b,c}</code>	Create several branches at once
<code>rmdir <dir></code>	Remove an <i>empty</i> directory
<code>rm <file></code>	Delete a file — permanently
<code>rm -r <dir></code>	Delete a directory and its contents
<code>rm -i <file></code>	Delete, asking first
<code>touch <file></code>	Create an empty file, or update timestamps
<code>touch a.txt b.txt c.txt</code>	Create several at once
<code>touch -d "2 days ago" f</code>	Set a specific timestamp
<code>cp <src> <dst></code>	Copy
<code>cp -r <dir> <dst></code>	Copy a directory tree
<code>mv <src> <dst></code>	Move, or rename
<code>cat <file></code>	Show the whole file
<code>cat a b > c</code>	Concatenate two files into a third
<code>tac <file></code>	Show it with the lines reversed
<code>head -n 5 <file></code>	First five lines
<code>tail -n 5 <file></code>	Last five lines
<code>tail -f <file></code>	Follow a file as it grows
<code>less <file></code>	Page through a long file
<code>wc -l <file></code>	Count lines
<code>find <dir> -name "*.tmp"</code>	Find files by name, recursively
<code>find . -type f wc -l</code>	Count files below here
<code>ln -s <target> <link></code>	Create a symbolic link

A.3 Wildcards

<code>*</code>	any characters, including none	<code>[abc]*</code>	starts with a, b or c
<code>?</code>	exactly one character	<code>[a-d]?</code>	a–d, then one char
<code>A*</code>	starts with A	<code>[!abc]*</code>	does not start with a, b or c
<code>A*txt</code>	starts A, ends txt	<code>[[:upper:]]*</code>	starts with a capital
<code>List??</code>	List plus two characters	<code>[[:digit:]]*</code>	starts with a digit
<code>A??f</code>	A, two characters, then f	<code>[![:digit:]]*</code>	does not start with a digit

A.4 Permissions and accounts

Command	Purpose
<code>chmod 755 <file></code>	Numeric mode: <code>rwX / r-X / r-x</code>
<code>chmod u+x <file></code>	Add execute for the owner
<code>chmod g-w <file></code>	Remove write from the group
<code>chmod u=rw,g=r,o= <file></code>	Set all nine bits explicitly
<code>chmod -R g+w <dir></code>	Apply through a tree
<code>chown user:group <file></code>	Change owner and group
<code>chgrp <group> <file></code>	Change the group only
<code>umask</code>	Show or set default permissions
<code>whoami</code>	Current username
<code>id</code>	UID, GID and group memberships
<code>groups</code>	Group names
<code>useradd -m -s /bin/bash u</code>	Create a user with home and shell
<code>passwd [user]</code>	Set a password
<code>usermod -aG <grp> <user></code>	Add to a supplementary group
<code>groupadd <group></code>	Create a group
<code>su - <user></code>	Switch user, full login
<code>sudo <command></code>	Run one command as root

A.4.1 Octal permission values

0	---	2	-w-	4	r--	6	rw-
1	--x	3	-wx	5	r-x	7	rwX

644	ordinary documents	600	private files, SSH keys
755	scripts, directories	700	private directories
664	shared, group-writable	444	read-only for everyone

A.5 Shell operators and redirection

Form	Meaning
<code>a ; b</code>	Run b afterwards, whatever happened to a
<code>a && b</code>	Run b only if a succeeded
<code>a b</code>	Run b only if a failed
<code>a b</code>	Send a's output into b
<code>cmd > file</code>	Output to file, replacing it
<code>cmd >> file</code>	Output appended to file
<code>cmd < file</code>	Input from file
<code>cmd 2> file</code>	Errors to file

Form	Meaning
<code>cmd > file 2>&1</code>	Both streams to file
<code>cmd &> file</code>	The same, more briefly
<code>cmd 2> /dev/null</code>	Discard errors
<code>cmd <<EOF ... EOF</code>	Here-document
<code>cmd &</code>	Run in the background
<code>\$(cmd)</code>	Substitute the command's output
<code>\$((expr))</code>	Arithmetic
<code>{1..5}</code>	Brace expansion

A.6 Bash scripting

Construct	Meaning
<code>#!/bin/bash</code>	Shebang; must be the first line
<code>chmod +x script.sh</code>	Make it executable
<code>./script.sh</code>	Run it
<code>source script.sh</code>	Run it in the current shell
<code>set -euo pipefail</code>	Fail fast on errors and unset variables
<code>\$0 \$1 \$ \$#</code>	Script name, first argument, all, count
<code>\$?</code>	Exit status of the last command
<code>\$RANDOM</code>	A random integer 0–32767
<code>if [...]; then ... fi</code>	Conditional
<code>-eq -ne -lt -le -gt -ge</code>	Numeric comparison
<code>= != -z -n</code>	String comparison
<code>-e -f -d -r -w -x</code>	File tests
<code>for x in list; do ... done</code>	Loop over a list
<code>while [...]; do ... done</code>	Loop while true
<code>case \$x in p) ;; esac</code>	Multi-way branch
<code>name() { ... }</code>	Define a function
<code>local var=value</code>	A variable scoped to a function
<code>bash -x script.sh</code>	Trace execution
<code>bash -n script.sh</code>	Check syntax only

A.7 Archiving and packages

Command	Purpose
<code>tar cvzf a.tar.gz dir/</code>	Create a gzipped archive
<code>tar tzf a.tar.gz</code>	List its contents
<code>tar xvzf a.tar.gz</code>	Extract it
<code>tar xvzf a.tar.gz -C /tmp</code>	Extract somewhere specific
<code>gzip / gunzip <file></code>	Compress / decompress in place
<code>zip -r a.zip dir/</code>	Create a ZIP
<code>unzip a.zip</code>	Extract a ZIP
<code>sudo apt update</code>	Refresh the package list
<code>sudo apt install <pkg></code>	Install
<code>sudo apt remove <pkg></code>	Remove
<code>dpkg -l / rpm -qa</code>	List installed packages

A.8 System and network

Command	Purpose
<code>sudo shutdown -h now</code>	Halt
<code>sudo shutdown -r now</code>	Reboot
<code>systemctl status <svc></code>	Is a service running
<code>systemctl start/stop</code>	Control it now
<code>systemctl enable/disable</code>	Control it at boot
<code>journalctl -u <svc></code>	That service's log
<code>ps aux</code>	Every process
<code>top</code>	Live process table
<code>kill <pid></code>	Ask a process to stop
<code>df -h</code>	Free disk space
<code>du -sh *</code>	Size of each item here
<code>free -h</code>	Memory in use
<code>uname -a</code>	Kernel and architecture
<code>uptime</code>	Load and time since boot
<code>ip a</code>	Network interfaces and addresses
<code>ping -c 4 <host></code>	Test reachability
<code>ss -tuln</code>	Listening ports
<code>sudo ufw status</code>	Firewall rules

A.9 Vim

Modes and files		Movement	
i	insert before cursor	h j k l	left, down, up, right
a	insert after cursor	w b e	word forward, back, end
o O	new line below, above	0 \$	start, end of line
v V	visual, visual line	gg G	first, last line
Ctrl+V	visual block	25G	line 25
Esc	back to Normal mode	%	matching bracket
:w	save	fx tx	to, before next x
:wq ZZ	save and quit	Ctrl+F	page down
:q! ZQ	quit, discarding changes	Ctrl+B	page up
Operators (+ motion or text object)		Text objects	
d	delete (cut)	iw aw	a word
y	yank (copy)	ip ap	a paragraph
c	change	i" a"	inside, around quotes
>> <<	indent, unindent	i (a (	inside, around brackets
gU gu	upper-, lowercase	it at	inside, around a tag
Editing		Search and replace	
x	delete a character	/text	search forward
dd	delete a line	?text	search backward
5dd	delete five lines	n N	next, previous match
yy	yank a line	*	search word under cursor
p P	paste after, before	:s/a/b/	replace, first on line
u	undo	:s/a/b/g	all on this line
Ctrl+R	redo	:%s/a/b/g	all in the file
.	repeat the last change	:%s/a/b/gc	all, with confirmation
Registers and macros		Buffers, windows, tabs	
"0p	paste the last yank	:ls	list buffers
"ayy	yank into register a	:b 2	go to buffer 2
"+y	yank to system clipboard	:sp :vsp	split horizontally, vertically
qa...q	record a macro	Ctrl+W then hjkl	move between windows
@a 20@a	replay it once, twenty times	gt gT	next, previous tab

A.10 Git

Command	Purpose
<code>git init</code>	Start tracking a folder
<code>git clone <url></code>	Copy a remote repository
<code>git config --global user.name</code>	Set your identity
<code>git status</code>	What state is everything in
<code>git add <file></code>	Stage a change
<code>git add .</code>	Stage everything
<code>git commit -m "msg"</code>	Take a snapshot
<code>git commit --amend</code>	Rewrite the last commit
<code>git log --oneline</code>	Compact history
<code>git diff</code>	Unstaged changes
<code>git diff --staged</code>	Staged changes
<code>git branch</code>	List branches
<code>git checkout -b <name></code>	Create a branch and switch to it
<code>git merge <name></code>	Merge a branch in
<code>git branch -d <name></code>	Delete a merged branch
<code>git remote add origin <url></code>	Register a remote
<code>git push -u origin main</code>	First push
<code>git pull</code>	Fetch and merge from the remote
<code>git restore <file></code>	Discard local changes
<code>git restore --staged f</code>	Unstage
<code>git revert <commit></code>	Safely reverse a commit
<code>git stash / git stash pop</code>	Set work aside and bring it back
<code>git reflog</code>	Find something you lost

A.11 SSH

Command	Purpose
<code>ssh user@host</code>	Connect
<code>ssh -p 2222 user@host</code>	Connect on another port
<code>ssh user@host "command"</code>	Run one remote command
<code>ssh -v user@host</code>	Verbose — for diagnosing failures
<code>ssh-keygen -t ed25519</code>	Generate a key pair
<code>ssh-copy-id user@host</code>	Install your public key on the server
<code>scp file user@host:/path</code>	Copy local to remote
<code>scp user@host:/path .</code>	Copy remote to local
<code>scp -r dir user@host:/p</code>	Copy a directory
<code>sftp user@host</code>	Interactive file transfer
<code>systemctl status ssh</code>	Is the server running
<code>ss -tuln grep :22</code>	Is anything listening on 22
<code>ssh-keygen -R <host></code>	Forget a changed host key

A.12 LaTeX

Command	Purpose
<code>latexmk -pdf file.tex</code>	Compile, as many passes as needed
<code>pdflatex file.tex</code>	One compilation pass
<code>bibtex file</code>	Process the bibliography
<code>\documentclass{article}</code>	Choose the document class
<code>\usepackage{graphicx}</code>	Load a package
<code>\maketitle</code>	Print the title block
<code>\tableofcontents</code>	Insert a contents list
<code>\section{}</code>	Sectioning
<code>\textbf{}</code> <code>\textit{}</code>	Bold, italic
<code>\begin{itemize}</code>	Bulleted list
<code>\begin{enumerate}</code>	Numbered list
<code>\$... \$</code>	Inline mathematics
<code>\begin{equation}</code>	Numbered display equation
<code>\frac{a}{b}</code>	Fraction
<code>\begin{tabular}</code>	Table body
<code>\includegraphics[width=]</code>	Insert an image
<code>\label{}</code> <code>\ref{}</code>	Cross-reference
<code>\cite{}</code>	Citation
<code>\bibliography{refs}</code>	Insert the bibliography

A.13 LibreOffice keyboard shortcuts

All modules		Module-specific	
Ctrl+S	Save	Ctrl+F12	Insert table (Writer)
Ctrl+Shift+S	Save As	Ctrl+F2	Function Wizard (Calc)
Ctrl+O	Open	Ctrl+1	Format Cells (Calc)
Ctrl+P	Print	Alt+=	AutoSum (Calc)
Ctrl+Z	Undo	Shift+F4	Cycle reference type (Calc)
Ctrl+Y	Redo	Ctrl+M	New slide (Impress)
Ctrl+F	Find and Replace	F5	Start slide show (Impress)
F11	Styles	Shift+F5	Show from current slide
Ctrl+F5	Sidebar	Ctrl+Shift+F5	Navigator (Impress)
Shift+F1	Extended tip	Esc	End slide show

A.14 Freeplane

Insert	New child node	Delete	Delete node and subtree
Enter	New sibling node	Ctrl+click	Add to selection
F2	Edit node text	Ctrl+L	Create a cross-link
Esc	Cancel editing	Ctrl+Shift+drag	Cross-link by dragging
Arrows	Move between nodes	Space	Collapse or expand

APPENDIX B

Going Further

These notes cover the syllabus. This appendix is about what comes after it — which of these tools reward more study, in what order, and where to go for each. Nothing here is examinable.

B.1 How to keep learning these tools

Three principles apply across everything in this module.

Read the local documentation first.

`man`, `--help` and `info` describe the exact version installed on your machine. A web search returns an answer for some other version, on some other distribution, possibly from 2011.

Build something you actually need.

A tutorial teaches the commands; a real task teaches when to use them. Keep your own notes in $\text{\LaTeX}$, version them with Git, back them up with `rsync` over SSH — one small project exercises four chapters of this book.

Learn to read errors properly.

Almost every tool here tells you what went wrong. `ssh -v`, `bash -x`, `git status` and the $\text{\LaTeX}$ log are diagnostic instruments, not noise.

B.2 LibreOffice

Next step	Why
Styles, seriously	The whole suite is built on them. A document formatted entirely with styles can be restyled in minutes; one formatted by hand cannot be restyled at all.
Calc: named ranges, pivot tables, data validation	These turn a spreadsheet from a calculator into something you can hand to someone else without it degrading.
Macros	Tools → Macros , in Basic or Python. The recorder writes a usable first draft of whatever you just did by hand.
Base and mail merge	When the data outgrows a spreadsheet. Also the point at which learning SQL starts paying for itself.

Where to look. The official user guides at <https://documentation.libreoffice.org> — one book per module, free, and far better than most commercial documentation. *Getting Started with LibreOffice* is the right one to read first.

B.3 Linux and the command line

Next step	Why
<code>find</code> and <code>xargs</code>	Select files by any property and act on them in bulk. This is the command that most expands what you can do in one line.
<code>grep</code> , <code>sed</code> , <code>awk</code>	The classic text-processing trio. Learn regular expressions once and all three become available at the same time.
<code>cron</code> and <code>systemd</code> timers	Scheduling. Combined with Chapter 12, this is how routine work stops being work.
Access control lists	<code>getfacl</code> , <code>setfacl</code> . When one owner and one group are not enough.
Inodes and links	Explains why permissions belong to a file rather than to its name, and what a hard link really is.
Your own virtual machine	VirtualBox and Ubuntu Server. A system you can break safely, and somewhere real to practise SSH.

Where to look. *The Linux Command Line* by William Shotts, free at <https://linuxcommand.org/tlcl.php>, covers Parts II and III of these notes in far greater depth and is the single best next book. `man hier` explains the whole filesystem layout in ten minutes.

B.4 Vim

Next step	Why
<code>vimtutor</code>	Thirty minutes, interactive, ships with Vim, and works on a throwaway copy you cannot damage. Do this before anything else.
<code>:help keyword</code>	The complete manual, offline, matching the version you actually have. <code>:help text-objects</code> and <code>:help registers</code> are good first queries.
Text objects and the dot command	The two features with the highest return per minute spent. Everything else in Chapter 9 is optional by comparison.
Macros over substitution	When an edit is too irregular for <code>:%s///</code> , record it once and replay it.
Vim keys everywhere	<code>set -o vi</code> in Bash, <code>less</code> , VS Code and JetBrains IDEs all offer Vim bindings — the investment is not confined to one editor.
Neovim	A compatible fork with a built-in terminal, Lua configuration and language-server support. Your <code>.vimrc</code> mostly carries across.

Where to look. `vimtutor` first, then `:help usr_toc` — the table of contents of the built-in user manual, which is a genuine book. Drew Neil's *Practical Vim* is the standard text for going from competent to fluent, and <https://vimhelp.org> is the same manual in a browser.

B.5 Bash

Next step	Why
<code>shellcheck</code>	A linter for shell scripts. Install it and run it on everything — it catches unquoted variables and portability problems, and explains each one.
<code>trap</code>	Clean up temporary files even when a script is interrupted.
<code>getopts</code>	Parse command-line options properly instead of by position.
<code>mktemp</code>	Safe temporary files.
Knowing when to stop	Past a few hundred lines, or as soon as you need real data structures, rewrite it in Python. Bash is a glue language; it is not a general-purpose one.

Where to look. `man bash` is the complete reference. The *Bash Guide* at <https://mywiki.woledge.org/BashGuide> is the best free tutorial, and the same wiki's *Bash Pitfalls* page is worth reading straight through — it is a catalogue of exactly the mistakes that produce scripts which work in testing and fail in use.

B.6 Git

Next step	Why
Deliberate conflicts	Create one on purpose and resolve it. Far less alarming once you have done it when nothing was at stake.
<code>git rebase</code>	Produces a linear history. Powerful, and dangerous on anything already pushed — learn the rule before the command.
Tags and releases	Marking a specific commit as a version.
Pull requests	How collaboration actually works on GitHub and GitLab.
<code>git bisect</code>	Binary search through history to find the commit that introduced a bug. Feels like magic the first time.

Where to look. *Pro Git* by Chacon and Straub, free at <https://git-scm.com/book>, is the standard reference and is genuinely readable. <https://learngitbranching.js.org> teaches branching interactively.

B.7 L^AT_EX

Next step	Why
Overleaf	https://www.overleaf.com . Browser-based, no installation, compiles as you type. The fastest way to start.
A local install	TeX Live plus VS Code with the LaTeX Workshop extension, when you want to work offline and under version control.
BibLaTeX and Biber	The modern replacement for BibTeX — better handling of non-English sources and far more citation styles.
TikZ	Drawing diagrams in the source itself. Every diagram in these notes is TikZ. Steep, but the results scale perfectly and live in version control alongside the text.
Beamer	Presentations in L ^A T _E X, when a deck needs to be reproducible or mathematics-heavy.

Where to look. *The Not So Short Introduction to L^AT_EX* is the standard free tutorial. <https://tex.stackexchange.com> has an answer for nearly everything else. The *TikZ and PGF* manual is enormous but its example gallery is the practical way in.

B.8 SSH

Next step	Why
Port forwarding	<code>-L</code> and <code>-R</code> tunnel other services through the encrypted channel — reaching a database that only listens on the server’s loopback interface, for instance.
<code>ssh-agent</code>	Holds your decrypted key in memory, so a passphrase-protected key needs the passphrase only once per session.
<code>tmux</code> or <code>screen</code>	Keeps a remote session alive across disconnections, so a long job survives your laptop closing.
<code>rsync</code>	Copies only what changed, resumes, and preserves permissions. Replaces <code>scp</code> for anything substantial.
<code>fail2ban</code>	Watches the authentication log and blocks addresses that keep guessing. Worth installing on any internet-facing server.

Where to look. `man ssh`, `man ssh_config` and `man sshd_config` are the authoritative references and are unusually well written. *SSH Mastery* by Michael Lucas is the short, practical book.

B.9 Freeplane and thinking tools

Next step	Why
Conditional styles	Format nodes automatically from their content or icons — every node tagged as a risk turns red without your touching it.
Filtering	Hide everything that does not match a condition, turning one large map into a series of focused views.
Scripting	Groovy, for bulk operations on large maps.
Knowing the limits	Mind maps are excellent for generating and organising ideas, and poor at strict sequence or formal logic. For those, use a flowchart, an outline, or a written document.

Where to look. The built-in help map — Freeplane documents itself as a mind map, which is also the best demonstration of what a large one looks like.

B.10 A suggested order

If you want to take one thing further, take it in roughly this order. Each step makes the next one easier.

#	Focus	Because
1	Command line fluency	Everything else in this list is faster from a terminal.
2	Git	Once your work is versioned, you can experiment without fear — which is what makes the rest safe to learn.
3	Bash scripting	Turns anything you have done twice into something you never do again.
4	SSH	Extends all of the above to machines that are not in front of you.
5	L ^A T _E X or a spreadsheet skill	Whichever matches the documents you actually produce.
6	A real programming language	Python, most likely. The point at which shell scripts stop being the right tool.

One last thing

The tools in this module are old — `vi`, Vim's ancestor, dates from 1976, T_EX from 1978, `tar` from 1979. They are still here because they were designed to compose with each other rather than to be complete on their own. That is the actual lesson of the module: small tools, each doing one thing, joined by pipes, files and plain text. Learn the joins and the individual tools become interchangeable.