Bash / Linux Shell Scripting

A Complete 12-Chapter Course

Topics covered:

Variables · I/O · Arithmetic · Conditionals · Loops · Functions

Arrays · Files & Text · Regex · Error Handling · Script Design

Exercises: 48 hands-on exercises with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 12

Topic 1 — Getting Started with Bash

🐚 Topic 1 — Getting Started with Bash

Before writing a single line of code, it helps to understand what Bash actually is, how it relates to your terminal, and the basic mechanics of creating and running a script. This chapter covers all of that — by the end you will have written and executed your first working script and understand exactly what happened when you ran it.

1 — What is the Shell?

When you open a terminal on a Linux system, you are not talking directly to the operating system kernel. Between you and the kernel sits a programme called a **shell** — a command interpreter that reads what you type, works out what you mean, and asks the kernel to carry it out.

THE TERMINAL

Window onto the shell

The terminal (or terminal emulator) is the window you see on screen. It handles display and keyboard input but has no intelligence of its own — it just passes what you type to the shell and displays what the shell sends back.

THE SHELL

The command interpreter

The shell reads your input, interprets it (expanding variables, glob patterns, redirections), and runs programmes or built-in commands. It also provides a complete programming language — loops, conditionals, functions, and variables.

THE KERNEL

The operating system core

The Linux kernel manages hardware, memory, processes, and the filesystem. The shell talks to the kernel on your behalf via **system calls**. As a script writer you rarely think about the kernel directly — that's the shell's job.

BASH

Bourne Again Shell

Bash (written in 1989 by Brian Fox) is the most widely used shell on Linux. It is the default shell on most Debian, Ubuntu, and Red Hat systems. The name is a pun on the original Bourne Shell (`sh`) that it replaced and extended.

Available Shells

SHELL	FULL NAME	NOTES
`bash`	Bourne Again Shell	*Default on most Linux distros. The focus of this course.*

SHELL	FULL NAME	NOTES
`sh`	Bourne Shell (or POSIX sh)	*The original. On modern systems `/bin/sh` is usually a link to dash or bash in POSIX mode. Fewer features than bash.*
`zsh`	Z Shell	*Default on macOS since Catalina. Very similar to bash with extra features.*
`fish`	Friendly Interactive Shell	*User-friendly with autosuggestions but not POSIX-compatible.*
`dash`	Debian Almquist Shell	*Lightweight and fast. Ubuntu uses it as `/bin/sh` for system scripts.*

Check which shell you are using

```
# Print your current shell
echo $SHELL
/bin/bash

# Print the exact version of bash installed
bash --version
GNU bash, version 5.2.21(1)-release (x86_64-pc-linux-gnu)

# List all shells installed on the system
cat /etc/shells
```

2 — What is a Shell Script?

A shell script is nothing more than a plain text file containing a sequence of commands — exactly the same commands you would type at the terminal, one per line. Instead of typing them one at a time, you write them all in a file and tell the shell to run the file. This lets you automate repetitive tasks, combine commands into reusable tools, and build complex workflows.

The simplest possible script

A script that prints a message and shows today's date. Save this as `hello.sh`.

```
#!/bin/bash
# My first bash script

echo "Hello, World!"
echo "Today is: "
date
```

That is a complete, working script. The following sections explain each part of it.

3 — The Shebang Line

The very first line of a script is special. The two characters `#!` (called a **shebang** or hashbang) followed by a path tell the operating system which interpreter to use to run this file.

```
#!/bin/bash
```

When the kernel sees a file starting with `#!`, it hands the file to the programme named after the `!`. In this case it runs `/bin/bash` and passes the script to it as input.

Common shebang lines

```
#!/bin/bash              # use bash explicitly
#!/usr/bin/env bash      # find bash via PATH (more portable)
#!/bin/sh                # use the system's POSIX shell (dash on Ubuntu)
#!/usr/bin/env python3   # same mechanism works for Python scripts
```

Which shebang should I use? Use `#!/bin/bash` when you want bash-specific features (arrays, `[[ ]]`, process substitution). Use `#!/usr/bin/env bash` when you need portability across systems where bash may not be in `/bin`. Use `#!/bin/sh` only when you intentionally want POSIX-only compatibility.

⚠ **No shebang = unpredictable behaviour.** Without a shebang line, the script will be run by your current shell — which might not be bash. Always include a shebang on the first line.

4 — Making a Script Executable

A newly created text file is not executable by default. Linux uses a permissions system to control who can read, write, and *execute* files. Before you can run a script with `./script.sh`, you must grant it execute permission.

Grant execute permission with chmod

```
# Create the script file
nano hello.sh

# Check the current permissions
ls -l hello.sh
-rw-r--r-- 1 philip philip 62 Jun  9 10:00 hello.sh
# rw-r--r-- = owner can read/write, others can only read. Nobody can execute.

# Grant execute permission to the owner
chmod +x hello.sh

# Verify the permission change
ls -l hello.sh
-rwxr-xr-x 1 philip philip 62 Jun  9 10:00 hello.sh
# The 'x' bits confirm execute permission is now set.
```

Understanding Permission Notation

CHARACTERS	WHO	MEANING
`rwx`	Owner (you)	Read, Write, Execute
`r-x`	Group	Read, no Write, Execute
`r-x`	Others (everyone else)	Read, no Write, Execute

> 💡 **chmod 755 vs chmod +x** — *`chmod +x` adds execute permission for everyone (owner, group, others). `chmod 755` does the same but also sets read/write for owner and read-only for everyone else. For personal scripts, `chmod +x` is fine. For scripts shared across users, `chmod 755` is more explicit.*

5 — Ways to Run a Script

There are three main ways to run a bash script, each with slightly different behaviour:

Three ways to run a script

```
# ── Method 1: Direct execution (requires chmod +x) ──────────────
./hello.sh
# The ./ means "in the current directory". The kernel reads the shebang
# and launches /bin/bash to run the script.

# ── Method 2: Call bash explicitly (no chmod needed) ────────────
bash hello.sh
# Tells bash directly to interpret the file. The shebang line is ignored
# (it becomes just a comment) because you specified the interpreter.

# ── Method 3: Source the script (dot command) ───────────────────
source hello.sh
# or equivalently:
. hello.sh
# Runs the script in the CURRENT shell session, not a new child process.
# Variables and functions defined in the script persist after it finishes.
# Useful for scripts that set environment variables (e.g. .bashrc).
```

METHOD	NEW PROCESS?	NEEDS CHMOD +X?	BEST USED FOR
`./script.sh`	Yes	Yes	Normal script execution
`bash script.sh`	Yes	No	Quick testing, debugging
`source script.sh`	No	No	Scripts that set variables/aliases

> **Why `./` ?** On Linux, the current directory is deliberately not included in the `PATH` for security reasons. Without `./`, the shell would search your `PATH` for a command named `hello.sh` and find nothing. The `./` prefix explicitly says "look in the current directory".

6 — Comments

A comment is text in a script that the shell ignores completely. Comments exist purely for the human reader. In bash, anything from a # character to the end of the line is a comment — **except** for the shebang on line 1.

Comment styles

```
#!/bin/bash
# ──────────────────────────────────────────────
# Script:  backup.sh
# Purpose: Creates a compressed backup of a directory
# Author:  Philip
# Date:    2026-06-09
# ──────────────────────────────────────────────

echo "Starting backup..."    # inline comment — comes after code

# The next line creates the archive
tar -czf backup.tar.gz /home/philip/documents
```

> 💡 **Good comments explain *why*, not *what*.** *The code already shows what is happening — a good comment explains the reasoning behind it. Instead of `# increment counter` (obvious), write `# skip the header line in the CSV` (explains purpose).*

7 — Script Structure and the PATH

Typical Script Layout

A well-structured script template

```
#!/bin/bash
# ──────────────────────────────────────────────
# Script name and one-line description
# ──────────────────────────────────────────────

# ── Configuration / constants ─────────────────
LOG_FILE="/var/log/myscript.log"
MAX_RETRIES=3

# ── Functions ─────────────────────────────────
greet() {
    echo "Hello, $1!"
}

# ── Main logic ────────────────────────────────
greet "World"
echo "Script complete."
```

Adding Scripts to your PATH

Once you have a collection of scripts you use regularly, you can place them in a directory and add that directory to your `PATH` so you can run them from anywhere without typing `./`.

Add ~/bin to your PATH

```
# Create a personal scripts directory
mkdir -p ~/bin

# Copy your script there
cp hello.sh ~/bin/hello

# Add ~/bin to PATH permanently by adding this line to ~/.bashrc
echo 'export PATH="$HOME/bin:$PATH"' >> ~/.bashrc

# Apply the change in the current session
source ~/.bashrc

# Now you can run the script from anywhere
hello
Hello, World!
```

Many modern Linux distributions already include ~/bin in PATH if the directory exists — check with `echo $PATH` first.

8 — A First Look at Debugging

Even in a simple script, things can go wrong. Bash has a built-in debug mode that prints each command before executing it — invaluable when a script isn't behaving as expected.

Running a script in debug mode

```
# Run with -x to see every command as it executes
bash -x hello.sh
+ echo 'Hello, World!'
Hello, World!
+ echo 'Today is: '
Today is:
+ date
Mon Jun  9 10:15:32 BST 2026

# Or add set -x inside the script to enable debug mode from that point
#!/bin/bash
set -x          # turn debug on
echo "hello"
set +x          # turn debug off
```

The + prefix in debug output marks commands executed by bash. We cover error handling and debugging in depth in Topic 11.

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `sysinfo.sh` that prints the following on separate lines: the current date and time, your username, your home directory, and the hostname of the machine.

Hint: the commands `date`, `whoami`, `echo $HOME`, and `hostname` will each give you one of the pieces of information you need.

▶ Show Sample Solution

EXERCISE 2

Write a script called `greet.sh` that accepts a name as a command-line argument and prints `Hello, [name]!`. If no argument is provided, it should print `Hello, World!` instead.

Hint: `$1` holds the first argument passed to the script. You can check whether it is empty with `if [ -z "$1" ]`.

▶ Show Sample Solution

EXERCISE 3

Create a script called `setup_project.sh` that creates a project directory structure. It should create a directory called `my_project` containing three subdirectories: `src`, `docs`, and `tests`. It should then print a confirmation message listing each directory created.

Hint: `mkdir -p` creates a directory and any missing parent directories in one command.

▶ Show Sample Solution

EXERCISE 4

Write a script called `mypath.sh` that prints each directory in your `PATH` environment variable on its own line, with a line number in front of each one. For example: `1: /usr/local/bin`, `2: /usr/bin`, etc.

Hint: `$PATH` contains directories separated by colons. You can split it by replacing `:` with newlines using `echo "$PATH" | tr ':' '\n'`. Piping through a loop with a counter will let you add line numbers.

▶ Show Sample Solution

CHAPTER 2 OF 12

Topic 2 — Variables and Data Types

📦 Topic 2 — Variables and Data Types

Variables are how a script remembers information. In this chapter you will learn how to create and use your own variables, how bash handles different kinds of data, and how to work with the special variables that bash sets up for you automatically — including the ones that tell you about command-line arguments and the exit status of commands.

1 — Creating and Using Variables

In bash you create a variable simply by assigning a value to a name. There is no `var` keyword, no type declaration — just a name, an equals sign, and a value.

Basic variable assignment and use

```
#!/bin/bash

# Assign values — NO spaces around the = sign
name="Philip"
city="London"
age=32

# Use a variable by prefixing its name with $
echo "Hello, my name is $name."
Hello, my name is Philip.

echo "I live in $city and I am $age years old."
I live in London and I am 32 years old.
```

⚠️ **No spaces around the = sign.** This is the most common beginner mistake. `name="Philip"` assigns a variable. `name = "Philip"` tries to run a command called `name` with arguments `=` and `"Philip"` — and fails with "command not found".

Curly Braces: ${variable}

You can wrap a variable name in curly braces — `${name}`. This is optional in most cases, but becomes *required* when the variable name is immediately followed by other characters that could be confused with part of the name.

When curly braces are necessary

```
fruit="apple"

# Without braces — bash reads this as variable 'fruitpie' (undefined)
echo "I want $fruitpie"
I want

# With braces — bash correctly reads 'fruit' and appends 'pie'
echo "I want ${fruit}pie"
I want applepie

# Also useful for clarity in complex strings
echo "${fruit}s are tasty"
apples are tasty
```

Variable Naming Rules

- Names may contain letters, digits, and underscores
- Names **must not start with a digit**
- Names are **case-sensitive** — `Name`, `name`, and `NAME` are three different variables
- By convention, **lowercase** for your own variables; **UPPERCASE** reserved for environment variables and constants

✓ VALID NAMES

```
username="philip"
file_count=10
_internal="ok"
MAX_RETRIES=3
myVar2="test"
```

✗ INVALID NAMES

```
2fast="no"       # starts with digit
my-var="no"      # hyphens not allowed
my var="no"      # spaces not allowed
$price="no"      # $ is for reading, not na
```

2 — Quotes and Variable Expansion

How you quote a value determines whether bash expands variables inside it. This is one of the most important distinctions to understand in bash.

Double quotes, single quotes, and no quotes

```
name="World"

# Double quotes — variables ARE expanded
echo "Hello, $name!"
Hello, World!

# Single quotes — variables are NOT expanded (everything is literal)
echo 'Hello, $name!'
Hello, $name!

# No quotes — variables expand BUT word splitting applies (avoid for strings)
echo Hello, $name!
Hello, World!

# Danger with no quotes: spaces in variables cause word splitting
file="my document.txt"
ls $file          # treated as two arguments: 'my' and 'document.txt'
ls "$file"        # correct: treated as one argument
```

💡 **Rule of thumb:** *Always double-quote variables when using them — `"$var"` — unless you have a specific reason not to. It prevents word-splitting and glob expansion from causing unexpected behaviour.*

Escaping Special Characters

🐧 Using a backslash to escape characters

```
# Inside double quotes, use \ to escape $ or " characters
price=5
echo "The cost is \$$price"
The cost is $5

# To include a literal double quote inside double quotes
echo "She said \"hello\""
She said "hello"

# To include a literal single quote inside single-quoted string — you can't.
# Instead, end the string, add an escaped quote, then resume:
echo 'it'\''s a trap'    # messy — double quotes are usually cleaner
```

3 — Data Types in Bash

Bash is a weakly typed language — all variables are stored as strings internally. However, bash can *treat* a variable as an integer when you use arithmetic operators, and you can use the `declare` built-in to give variables explicit attributes.

TYPE / ATTRIBUTE	HOW TO CREATE	BEHAVIOUR
String (default)	`name="hello"`	*Everything is a string unless you specify otherwise. Arithmetic on an unset variable returns 0.*
Integer	`declare -i count=5`	*Bash enforces integer-only values. Assigning a non-integer sets the variable to 0. Arithmetic is performed automatically.*
Read-only	`readonly MAX=100` or `declare -r MAX=100`	*Value cannot be changed after declaration. Attempting to reassign produces an error.*
Exported (env var)	`export VAR="value"` or `declare -x VAR`	*Variable is passed to child processes (subshells, scripts called from this script).*
Array	`declare -a items`	*Indexed array. Covered in Topic 8.*
Associative array	`declare -A map`	*Key-value map. Also covered in Topic 8.*

declare and readonly examples

```
# Integer variable — arithmetic assigned directly
declare -i count=10
count+=5
echo "$count"
15

# Read-only — cannot be reassigned
readonly MAX_SIZE=100
MAX_SIZE=200
bash: MAX_SIZE: readonly variable

# Check variable attributes with declare -p
declare -p count
declare -i count="15"
declare -p MAX_SIZE
declare -r MAX_SIZE="100"
```

4 — Environment Variables

Environment variables are variables that are passed down from a parent process to its child processes. Your shell session already has dozens of them set before you run a single script — they describe the system, your user account, and your preferences.

VARIABLE	CONTAINS	EXAMPLE VALUE
`$HOME`	Your home directory	*/home/philip*
`$USER`	Your username	*philip*
`$PATH`	Colon-separated list of directories searched for executables	*/usr/local/bin:/usr/bin:/bin*
`$PWD`	Current working directory	*/home/philip/scripts*
`$OLDPWD`	Previous working directory	*/home/philip*
`$SHELL`	Path to the current shell	*/bin/bash*
`$HOSTNAME`	Machine hostname	*raspberrypi*
`$LANG`	Language / locale setting	*en_GB.UTF-8*
`$EDITOR`	Default text editor	*nano*
`$TERM`	Terminal type	*xterm-256color*

Creating and exporting your own environment variables

```
# A variable created normally is LOCAL — not visible to child processes
greeting="hello"
bash -c 'echo $greeting'
                        # empty — child process can't see it

# export makes it available to child processes
export greeting="hello"
bash -c 'echo $greeting'
hello

# List all current environment variables
env
# or just the ones matching a pattern
env | grep "HOME"
HOME=/home/philip
```

Changes made with `export` inside a script only affect that script and its children — never the parent shell that launched the script.

5 — Special Variables

Bash automatically populates a set of read-only variables that give you information about the script itself, its arguments, and the outcome of the last command. These are among the most useful variables in shell scripting.

Positional Parameters — Script Arguments

VARIABLE	CONTAINS
`$0`	The name of the script itself (as it was called)
`$1` … `$9`	The 1st through 9th arguments passed to the script
`${10}` … `${N}`	Arguments beyond the 9th (must use curly braces)
`$#`	The total number of arguments passed
`$@`	All arguments as separate quoted strings — `"$1" "$2" "$3" …`
`$*`	All arguments as a single string — usually less useful than `$@`

Positional parameters in action

```
#!/bin/bash
# args_demo.sh

echo "Script name : $0"
echo "First arg   : $1"
echo "Second arg  : $2"
echo "All args    : $@"
echo "Arg count   : $#"
```

```
# Running the script:
./args_demo.sh hello world
Script name : ./args_demo.sh
First arg   : hello
Second arg  : world
All args    : hello world
Arg count   : 2
```

Process and Status Variables

VARIABLE	CONTAINS
`$?`	Exit status of the last command (0 = success, non-zero = failure)
`$$`	PID (Process ID) of the current script
`$!`	PID of the last background command (started with `&`)
`$-`	Current shell option flags

Using $? to check command success

```
# $? holds the exit status of the LAST command that ran
ls /home
echo "Exit status: $?"
Exit status: 0           # 0 means success

ls /nonexistent_directory
ls: cannot access '/nonexistent_directory': No such file or directory
echo "Exit status: $?"
Exit status: 2           # non-zero means failure

# Practical use: check if the last command succeeded
cp file.txt backup.txt
if [ "$?" -eq 0 ]; then
    echo "Backup created successfully."
else
    echo "Backup FAILED."
fi
```

Read $? immediately after the command — if you run any other command first (even echo), $? will reflect that command's status instead.

**$@ vs $*** — The difference only matters when they are double-quoted. "$@" expands to "$1" "$2" "$3" (each argument properly quoted separately). "$*" expands to "$1 $2 $3" (all arguments joined into one string). Use "$@" when passing arguments to another command — it preserves arguments that contain spaces.

6 — Command Substitution

Command substitution lets you capture the output of a command and use it as a value — either assigning it to a variable or inserting it directly into a string.

Two syntaxes for command substitution

```
# Modern syntax (recommended): $( )
today=$(date +%Y-%m-%d)
echo "Today is $today"
Today is 2026-06-09

# Can be used directly inside a string
echo "You are logged in as $(whoami) on $(hostname)"
You are logged in as philip on raspberrypi

# Can be nested
parent_dir=$(dirname $(pwd))
echo "Parent: $parent_dir"

# Old backtick syntax — still works but harder to nest and read
files=`ls -1 | wc -l`
echo "Files in directory: $files"
```

Always use $( ) rather than backticks for new code. Backtick syntax is harder to read and cannot be nested without escaping.

7 — Default Values and Unsetting Variables

Bash provides a compact syntax for supplying default values when a variable is not set or is empty. This is far cleaner than writing an `if` check every time.

Parameter expansion for defaults

```
# ${var:-default} — use default if var is unset or empty
colour=""
echo "Colour: ${colour:-blue}"
Colour: blue
# Note: colour is still empty after this — the default is only used in the expansion

# ${var:=default} — use default AND assign it to var if unset or empty
echo "Colour: ${colour:=blue}"
Colour: blue
echo "$colour"
blue
# Now colour has been set to "blue"

# ${var:?message} — print an error message and exit if var is unset
required=""
echo "${required:?Error: required variable is not set}"
bash: required: Error: required variable is not set

# ${var:+replacement} — use replacement only if var IS set
debug="true"
echo "${debug:+[DEBUG MODE ON]}"
[DEBUG MODE ON]
```

Unsetting a variable

```
temp="temporary value"
echo "$temp"
temporary value

# Remove the variable entirely
unset temp
echo "'$temp'"
''
# $temp is now completely unset (not just empty)

# Check if a variable is set
if [ -z "${temp+x}" ]; then
    echo "temp is not set"
fi
```

8 — Quick Reference

SYNTAX	WHAT IT DOES
`name="value"`	Assign a variable (no spaces around =)
`$name` / `${name}`	Read a variable's value
`"$name"`	Read variable with word-splitting protection (always prefer this)
`'$name'`	Literal string — no expansion
`export name`	Make variable available to child processes
`readonly name`	Prevent variable from being reassigned
`unset name`	Remove a variable entirely
`$(command)`	Command substitution — capture output of a command
`${var:-default}`	Use default if var is empty/unset
`${var:=default}`	Use default AND assign it if var is empty/unset
`${var:?msg}`	Exit with error message if var is empty/unset
`$0`	Script name
`$1` … `$9`	Positional arguments
`$#`	Number of arguments
`$@`	All arguments (individually quoted)
`$?`	Exit status of last command
`$$`	PID of current script

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `profile.sh` that stores your name, age, and favourite programming language in variables, then prints a short bio using those variables. All three pieces of data should appear in a single `echo` statement.

Hint: assign each value to a descriptive variable name, then use all three variables inside one double-quoted string.

▶ Show Sample Solution

EXERCISE 2

Write a script called `args_info.sh` that prints: the script's own name, how many arguments were passed, the first and second argument (or the text "not provided" if they were not given), and all arguments on one line.

Hint: use `$0`, `$#`, `${1:-not provided}`, `${2:-not provided}`, and `$@`. Test it by running it with no arguments, one argument, and two arguments.

▶ Show Sample Solution

EXERCISE 3

Write a script called `cmd_check.sh` that runs the command `ls /tmp` and then `ls /nonexistent`, printing the exit status after each one with a human-readable label ("Success" or "Failed").

Hint: capture `$?` immediately after each command. You can use `if [ "$status" -eq 0 ]` to test it. Store the exit status in a variable first so it doesn't get overwritten.

▶ Show Sample Solution

EXERCISE 4

Write a script called `snapshot.sh` that captures the current date, the current user, the current directory, and the number of files in the current directory into variables using command substitution, then prints a formatted summary. Run it from a couple of different directories to verify it works correctly each time.

Hint: use `$(date +"%Y-%m-%d %H:%M")`, `$(whoami)`, `$(pwd)`, and `$(ls -1 | wc -l)` to populate your variables.

▶ Show Sample Solution

CHAPTER 3 OF 12

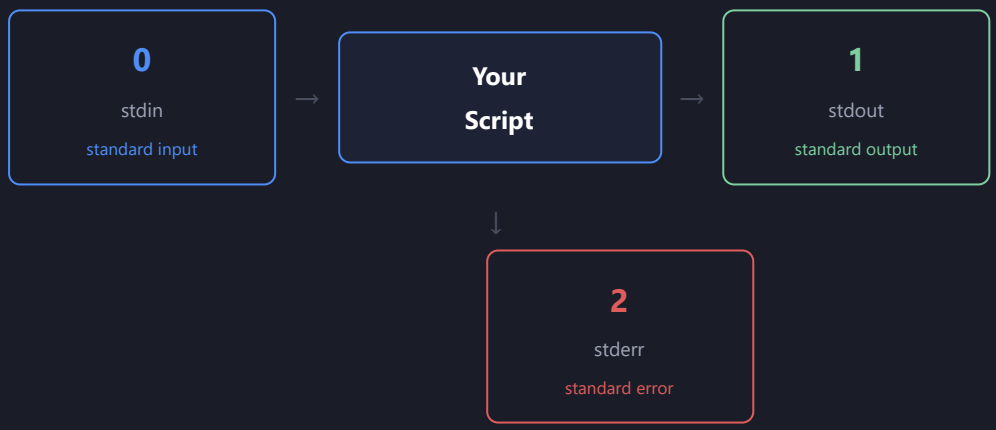
Topic 3 — Input and Output

📡 Topic 3 — Input and Output

Almost everything a script does involves reading something in or writing something out. This chapter covers the full toolkit: `echo` and `printf` for output, `read` for interactive user input, the redirection operators that send data to files, and pipes that chain commands together. By the end you will also understand how here-documents let you embed multi-line input directly inside a script.

1 — Standard Streams

Every process on Linux has three standard data streams automatically connected when it starts. Understanding them is the key to understanding redirection and pipes.



STREAM	FD	DEFAULT CONNECTION	USED FOR
`stdin`	0	Keyboard	Input that the script reads
`stdout`	1	Terminal screen	Normal output (`echo` , `printf`)
`stderr`	2	Terminal screen	Error messages — separate from stdout so errors can be handled independently

Redirection operators change where these streams connect — to files, to other streams, or to other commands via pipes.

2 — Output with echo

`echo` is the simplest way to print to stdout. It outputs its arguments followed by a newline.

echo options

```
# Basic output — adds a newline at the end
echo "Hello, World!"
Hello, World!

# -n suppresses the trailing newline
echo -n "Enter your name: "
# Cursor stays on the same line — useful before read

# -e enables interpretation of escape sequences
echo -e "Line one\nLine two\nLine three"
Line one
Line two
Line three

echo -e "Column 1\tColumn 2\tColumn 3"
Column 1        Column 2        Column 3

# Useful escape sequences with -e
# \n  — newline    \t  — tab    \\  — backslash
# \a  — alert bell \b  — backspace
```

The behaviour of `echo` with no flags varies slightly between systems. For consistent formatted output across platforms, use `printf` (see section 3).

Writing to stderr

By convention, error messages should go to stderr (file descriptor 2), not stdout. This lets the caller separate normal output from errors.

Sending output to stderr

```
# Redirect echo's output to stderr using >&2
echo "ERROR: File not found." >&2

# Practical pattern: write a reusable error function
error() {
    echo "ERROR: $1" >&2
}

error "Could not read config file."
ERROR: Could not read config file.   # printed to stderr
```

3 — Formatted Output with printf

`printf` gives you precise control over formatting. It works like C's `printf`: a format string with placeholders, followed by the values to insert. Unlike `echo`, it does *not* add a newline automatically — you must include `\n` explicitly.

printf format specifiers

```
# %s — string, %d — integer, %f — floating point
printf "Hello, %s!\n" "Philip"
Hello, Philip!

printf "You have %d messages.\n" 42
You have 42 messages.

printf "Price: %.2f\n" 9.5
Price: 9.50

# Width and alignment — great for building tables
# %-20s — left-align in a 20-char column
# %8d   — right-align integer in 8-char column
printf "%-20s %8s %10s\n" "Name" "Age" "City"
printf "%-20s %8d %10s\n" "Philip"  32 "London"
printf "%-20s %8d %10s\n" "Anna"    28 "Budapest"
printf "%-20s %8d %10s\n" "Kenji"   35 "Tokyo"
Name                        Age       City
Philip                       32     London
Anna                         28   Budapest
Kenji                        35      Tokyo

# Store formatted output in a variable
line=$(printf "%-20s %5d" "count" 99)
echo "$line"
```

SPECIFIER	TYPE	EXAMPLE
`%s`	String	`printf "%s" "hello"` → `hello`
`%d`	Integer (decimal)	`printf "%d" 42` → `42`
`%f`	Floating point	`printf "%.2f" 3.14159` → `3.14`
`%05d`	Zero-padded integer	`printf "%05d" 7` → `00007`
`%-10s`	Left-aligned string, 10 chars wide	`printf "%-10s|" "hi"` → `hi |`
`%10s`	Right-aligned string, 10 chars wide	`printf "%10s|" "hi"` → `hi|`
`\n`	Newline	*Must be explicit — printf does not add one automatically*
`\t`	Tab	

4 — Reading User Input with read

The `read` built-in reads a line from stdin and stores it in one or more variables. It is the standard way to make an interactive script that prompts the user for information.

Basic read usage

```
#!/bin/bash

# Basic: prompt then read
echo -n "What is your name? "
read name
echo "Hello, $name!"

# -p: inline prompt (cleaner — no need for a separate echo)
read -p "Enter your city: " city
echo "You live in $city."

# Read multiple variables — words split on whitespace
read -p "Enter first and last name: " first last
echo "First: $first  Last: $last"

# If more words than variables, the last variable gets the remainder
# read first last  → "John Paul Jones" gives first=John last="Paul Jones"
```

Useful read Options

read flags

```
# -s: silent mode — input is not echoed (for passwords)
read -s -p "Password: " password
echo   # print newline after hidden input
echo "Password stored (not shown)."

# -n: read exactly N characters (no Enter needed)
read -n 1 -p "Press any key to continue..."
echo

# -t: timeout in seconds — returns non-zero exit if time expires
read -t 5 -p "You have 5 seconds to answer: " answer
if [ "$?" -ne 0 ]; then
    echo "\nTime's up!"
fi

# -r: raw mode — backslash is NOT treated as an escape character
# Always use -r when reading file paths or arbitrary input
read -r -p "Enter a file path: " filepath

# -a: read words into an array
read -r -a colours -p "Enter colours: "
echo "First colour: ${colours[0]}"
```

Always use `read -r` as the default — without it, a backslash at the end of a line acts as a line continuation, which can cause silent data loss.

> 💡 **Reading from a file line by line** — *the most common use of `read` beyond interactive input is reading a file line by line in a loop: `while IFS= read -r line; do echo "$line"; done < file.txt`. The `IFS=` prevents leading/trailing whitespace from being stripped. This pattern is covered in depth in Topic 6 (Loops).*

5 — Redirection

Redirection operators change where a command's stdin, stdout, or stderr is connected. Instead of the terminal, you can send output to a file, read input from a file, or route error messages separately.

Output Redirection

Writing output to files

```
# > creates (or overwrites) a file with stdout
echo "Hello" > output.txt
cat output.txt
Hello

# >> appends to a file (does not overwrite)
echo "World" >> output.txt
cat output.txt
Hello
World

# 2> redirects stderr to a file
ls /nonexistent 2> errors.log
cat errors.log
ls: cannot access '/nonexistent': No such file or directory

# 2>> appends stderr to a file
ls /another_bad_path 2>> errors.log

# &> (or >&) redirects both stdout AND stderr to a file
./my_script.sh &> all_output.log
```

Input Redirection

Reading input from a file

```
# < feeds a file into a command's stdin
sort < names.txt
# same as: sort names.txt  (for commands that accept file arguments)
# but < works universally for any command that reads stdin

# Useful when a command does not accept a filename argument
while read -r line; do
    echo "Line: $line"
done < data.txt
```

Combining Redirections

Separating and merging stdout and stderr

```
# Send stdout to one file, stderr to another
./script.sh > output.log 2> errors.log

# Redirect stderr to the same place as stdout (order matters!)
./script.sh > all.log 2>&1
# Read as: stdout → all.log, then stderr → wherever stdout now points

# Common mistake — reversed order sends stderr to the OLD stdout (terminal)
./script.sh 2>&1 > all.log    # WRONG: stderr still goes to terminal

# Discard all output (send to /dev/null — the black hole)
./script.sh >/dev/null 2>&1

# Discard only errors
./script.sh 2>/dev/null
```

`/dev/null` is a special device that discards anything written to it and returns EOF when read. It is the standard way to suppress output you don't care about.

OPERATOR	EFFECT
`cmd > file`	Write stdout to file (overwrite)
`cmd >> file`	Append stdout to file
`cmd < file`	Read stdin from file
`cmd 2> file`	Write stderr to file (overwrite)
`cmd 2>> file`	Append stderr to file
`cmd &> file`	Write both stdout and stderr to file
`cmd > file 2>&1`	Write both to file (POSIX-compatible form)
`cmd 2>/dev/null`	Discard all error output
`cmd >/dev/null 2>&1`	Discard all output entirely

6 — Pipes

A pipe | connects the stdout of one command directly to the stdin of the next, letting you chain commands together into a processing pipeline. No intermediate file is needed — data flows in memory.

Building command pipelines

```
# Count lines in a file
cat names.txt | wc -l

# Sort a file, remove duplicates, show the first 5
cat names.txt | sort | uniq | head -5

# Find all running bash processes
ps aux | grep "bash" | grep -v "grep"

# Count how many lines contain the word "error" (case-insensitive)
cat app.log | grep -i "error" | wc -l

# Convert a list of filenames to uppercase
ls | tr '[:lower:]' '[:upper:]'
```

The exit status of a pipeline is the exit status of its last command. To catch failures in earlier commands, use `set -o pipefail` (covered in Topic 11).

tee — Branch a Pipeline

The `tee` command reads stdin and writes it to *both* stdout and a file simultaneously — like a T-junction in a pipe. Useful when you want to log output and still see it on screen.

Using tee to log and display at the same time

```
# Display output on screen AND save to a file
./build.sh | tee build.log

# Append to the file instead of overwriting
./test.sh | tee -a test.log

# Capture both stdout and stderr, display and log
./script.sh 2>&1 | tee all.log
```

7 — Here-Documents

A here-document (heredoc) lets you embed a block of multi-line text directly in a script and feed it as stdin to a command. This is far cleaner than running many `echo` statements in a row.

Basic here-document syntax

```
#!/bin/bash

# The delimiter (EOF here, but any word works) marks the start and end
cat <<EOF
This is line one.
This is line two.
Today is $(date +%Y-%m-%d) and the user is $USER.
EOF
This is line one.
This is line two.
Today is 2026-06-09 and the user is philip.

# Write a multi-line file in one block
cat <<EOF > config.txt
host=localhost
port=8080
debug=false
EOF

# Suppress variable expansion with a quoted delimiter
cat <<'EOF'
The variable $USER will not be expanded here.
This is printed literally.
EOF
The variable $USER will not be expanded here.

# Indent the closing delimiter with <<- (strips leading TABS, not spaces)
if true; then
    cat <<-EOF
        This heredoc is indented with tabs.
        The leading tabs are stripped from output.
        EOF
fi
```

The closing delimiter must appear on a line by itself with no leading spaces (unless using <<- with tabs). A common source of "unexpected EOF" errors.

Practical here-doc: generating a report file

Here-docs are ideal for generating config files, email bodies, or HTML fragments from within a script.

```
#!/bin/bash
report_file="report_$(date +%Y%m%d).txt"

cat <<EOF > "$report_file"
========================================
  System Report — $(date)
========================================
Host   : $(hostname)
User   : $USER
Uptime : $(uptime -p)
Disk   : $(df -h / | tail -1 | awk '{print $5 " used"}')
========================================
EOF

echo "Report saved to $report_file"
```

Here-Strings

A here-string <<< is a compact way to pass a single string as stdin to a command — without a file or a full heredoc.

Here-string examples

```
# Feed a string to grep without needing echo | grep
grep "World" <<< "Hello, World!"
Hello, World!

# Useful with read to parse a string into variables
csv_line="Philip,32,London"
IFS=',' read -r name age city <<< "$csv_line"
echo "Name: $name  Age: $age  City: $city"
Name: Philip  Age: 32  City: London
```

8 — Quick Reference

COMMAND / SYNTAX	WHAT IT DOES
`echo "text"`	Print text with a trailing newline
`echo -n "text"`	Print without trailing newline

COMMAND / SYNTAX	WHAT IT DOES
`echo -e "a\nb"`	Print with escape sequences interpreted
`echo "msg" >&2`	Print to stderr
`printf "%s\n" "text"`	Formatted print (no automatic newline)
`read -r var`	Read a line from stdin into var
`read -r -p "prompt" var`	Prompt then read
`read -r -s -p "pw: " pw`	Read silently (password)
`read -r -t 5 var`	Read with 5-second timeout
`cmd > file`	Redirect stdout to file (overwrite)
`cmd >> file`	Append stdout to file
`cmd 2> file`	Redirect stderr to file
`cmd > f 2>&1`	Redirect stdout and stderr to file
`cmd >/dev/null`	Discard output
`cmd1 | cmd2`	Pipe stdout of cmd1 to stdin of cmd2
`cmd | tee file`	Display output AND save to file
`cmd <<EOF … EOF`	Here-document: feed block of text as stdin
`cmd <<< "string"`	Here-string: feed single string as stdin

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write an interactive script called `register.sh` that asks the user for their first name, last name, and age (each on a separate prompt), then prints a formatted summary. The age prompt should use `read -t 10` — if the user doesn't respond in 10 seconds, print "No age given" and continue.

Hint: use `read -r -p` for the name prompts, `read -r -t 10 -p` for the age prompt, and check `$?` after the age read to detect a timeout.

▶ Show Sample Solution

EXERCISE 2

Write a script called `logger.sh` that accepts a message as a command-line argument, writes it (with a timestamp) to a file called `app.log`, and also prints it to the screen. If no argument is given, write "ERROR: no message provided" to stderr and exit. Run it several times to verify it appends rather than overwrites.

Hint: use `${1:?...}` or an explicit `if [ -z "$1" ]` check, `>>` to append to the log file, and `tee -a` to show and log simultaneously.

▶ Show Sample Solution

EXERCISE 3

Write a script called `table.sh` that uses `printf` to print a neatly aligned table of at least four items with three columns: Name, Price (formatted to 2 decimal places), and In Stock (Yes/No). Include a header row with a separator line made of dashes.

Hint: use `printf "%-20s %8s %10s\n"` for the header and `printf "%-20s %8.2f %10s\n"` for the data rows. Generate the separator line with `printf '%0.s-' {1..42}` or a hardcoded string.

▶ Show Sample Solution

EXERCISE 4

Write a script called `gen_config.sh` that uses a here-document to generate a configuration file called `server.conf`. The file should include the current hostname, current date, and a fixed set of configuration values. Also redirect any errors from the file-write to a file called `gen_config.err`.

Hint: use `cat <<EOF > server.conf 2> gen_config.err`. Include at least one `$(command)` substitution inside the heredoc to embed live system values.

▶ Show Sample Solution

CHAPTER 4 OF 12

Topic 4 — Arithmetic and String Operations

🔢 Topic 4 — Arithmetic and String Operations

Bash stores everything as a string, but it can perform integer arithmetic natively and provides a rich set of parameter expansion operators for slicing, replacing, and transforming strings — all without calling an external command. This chapter covers the arithmetic context `$(( ))`, floating-point math with `bc`, and the full string manipulation toolkit built into bash's parameter expansion syntax.

1 — Integer Arithmetic with $(( ))

The arithmetic expansion `$(( expression ))` evaluates an integer expression and substitutes the result. It is the standard, preferred way to do arithmetic in bash — fast, built-in, and readable.

Basic arithmetic operations

```
#!/bin/bash

a=10
b=3

echo "Addition       : $((a + b))"      → 13
echo "Subtraction    : $((a - b))"      → 7
echo "Multiplication : $((a * b))"      → 30
echo "Division       : $((a / b))"      → 3   (integer — truncates)
echo "Modulo         : $((a % b))"      → 1
echo "Exponentiation : $((a ** b))"     → 1000

# Store result in a variable
result=$(( a * b + 5 ))
echo "Result: $result"
Result: 35

# Variables inside (( )) do NOT need the $ prefix
total=$(( a + b ))      # both work
total=$(( $a + $b ))    # also fine, but redundant
```

Increment, Decrement, and Compound Assignment

Updating variables in place

```
count=0

# Increment by 1
count=$(( count + 1 ))     # explicit form
$(( count++ ))               # post-increment (returns old value, then adds 1)
$(( ++count ))               # pre-increment  (adds 1 first, then returns)
(( count++ ))                # (( )) alone — no $ needed when not capturing value

# Compound assignment operators
n=10
(( n +=  5  ))  # n = n + 5  → 15
(( n -=  3  ))  # n = n - 3  → 12
(( n *=  2  ))  # n = n * 2  → 24
(( n /=  4  ))  # n = n / 4  → 6
(( n %=  4  ))  # n = n % 4  → 2
echo "$n"
2
```

((expr)) without the leading $ evaluates the expression for its side effects (like updating a counter) and sets the exit status to 0 (true) if the result is non-zero, 1 (false) if zero. This is useful in loop conditions (Topic 6).

Arithmetic with let

The `let` built-in is an older alternative that evaluates arithmetic expressions without needing `$(( ))` syntax. It is less commonly used in modern scripts but still appears in legacy code.

let vs $(( ))

```
# let evaluates the expression directly
let x=5+3
echo "$x"
8

let x++
echo "$x"
9

# Equivalent using $(( )) — preferred in modern scripts
x=$(( 5 + 3 ))
(( x++ ))
```

2 — Floating-Point Arithmetic with bc

Bash's built-in arithmetic only handles integers. For decimal calculations you need an external tool — `bc` (basic calculator) is the standard choice. You pipe an expression to it as a string and capture the result.

Using bc for decimal maths

```
# Basic: pipe an expression string to bc
echo "3.14 * 2" | bc
6.28

# scale= controls decimal places
echo "scale=2; 10 / 3" | bc
3.33

echo "scale=4; sqrt(2)" | bc -l    # -l loads the math library (sqrt, sin, cos...)
1.4142

# Store result in a variable using command substitution
price=49.99
tax_rate=0.20
total=$(echo "scale=2; $price * (1 + $tax_rate)" | bc)
echo "Total with tax: £$total"
Total with tax: £59.98

# Comparison — bc returns 1 (true) or 0 (false)
result=$(echo "3.14 > 3" | bc)
if [ "$result" -eq 1 ]; then
    echo "3.14 is greater than 3"
fi
```

bc is an external programme — it adds a small overhead per call. For scripts doing thousands of float calculations, consider Python or awk instead.

bc -l math functions: the `-l` flag loads bc's standard maths library, which provides `sqrt(x)`, `s(x)` (sine), `c(x)` (cosine), `a(x)` (arctangent), `e(x)` (e^x), and `l(x)` (natural log). It also sets the default scale to 20 decimal places.

3 — String Length and Slicing

Bash provides parameter expansion operators for extracting information from strings. These work without any external commands — they are built directly into the shell.

String length and substrings

```
str="Hello, World!"

# ${#var} — length of string
echo "Length: ${#str}"
Length: 13

# ${var:offset} — substring from offset to end
echo "${str:7}"
World!

# ${var:offset:length} — substring of given length
echo "${str:0:5}"
Hello

echo "${str:7:5}"
World

# Negative offset — count from the END of the string
# Note: space before negative number avoids confusion with ${var:-default}
echo "${str: -6}"
World!

echo "${str: -6:5}"
World
```

4 — Prefix and Suffix Removal

These operators strip matching patterns from the beginning or end of a string. They are extremely useful for manipulating file paths, extensions, and structured strings — all without calling `sed` or `cut`.

 # ## % %% operators

```
filepath="/home/philip/documents/report.final.txt"

# ${var#pattern}  — remove SHORTEST match from the FRONT
echo "${filepath#*/}"
home/philip/documents/report.final.txt

# ${var##pattern} — remove LONGEST match from the FRONT
echo "${filepath##*/}"          # strips everything up to last /
report.final.txt

# ${var%pattern}  — remove SHORTEST match from the END
echo "${filepath%.*}"           # strips last extension
/home/philip/documents/report.final

# ${var%%pattern} — remove LONGEST match from the END
echo "${filepath%%.*}"          # strips everything from first dot
/home/philip/documents/report
```

Practical: extract components from a file path

These four operators handle the most common path manipulation tasks without needing basename or dirname.

```
file="/var/log/nginx/access.log"

# Filename only (equivalent to basename)
filename="${file##*/}"
echo "Filename : $filename"
Filename : access.log

# Directory only (equivalent to dirname)
dir="${file%/*}"
echo "Directory: $dir"
Directory: /var/log/nginx

# Extension only
ext="${filename##*.}"
echo "Extension: $ext"
Extension: log

# Filename without extension
base="${filename%.*}"
echo "Base name: $base"
Base name: access
```

> **Pattern syntax:** The patterns in #, ##, %, %% use glob wildcards, not regular expressions. * matches any sequence of characters, ? matches a single character, and [abc] matches a character class. Regular expression matching is covered in Topic 10.

5 — Search and Replace

Bash can search for a pattern in a string and replace it — again using parameter expansion, with no external tools.

/ and // replacement operators

```
sentence="the cat sat on the mat"

# ${var/pattern/replacement} — replace FIRST occurrence
echo "${sentence/the/a}"
a cat sat on the mat

# ${var//pattern/replacement} — replace ALL occurrences
echo "${sentence//the/a}"
a cat sat on a mat

# ${var/pattern/} — delete pattern (replace with nothing)
echo "${sentence// /}"          # remove all spaces
thecatsatonthemat

# Replace only at the start (# anchor)
echo "${sentence/#the/a}"
a cat sat on the mat

# Replace only at the end (% anchor)
echo "${sentence/%mat/rug}"
the cat sat on the rug

# Practical: replace spaces with underscores in a filename
name="my document file.txt"
safe_name="${name// /_}"
echo "$safe_name"
my_document_file.txt
```

6 — Case Conversion

Bash 4.0 introduced built-in case conversion operators. These are available on almost all modern Linux systems (check with `bash --version` — you need 4.0 or higher).

Uppercase and lowercase operators (bash 4+)

```
str="Hello, World!"

# ${var^^} — convert ALL characters to UPPERCASE
echo "${str^^}"
HELLO, WORLD!

# ${var,,} — convert ALL characters to lowercase
echo "${str,,}"
hello, world!

# ${var^} — capitalise FIRST character only
word="hello"
echo "${word^}"
Hello

# ${var,} — lowercase FIRST character only
word="HELLO"
echo "${word,}"
hELLO

# With a pattern — only matching characters are changed
echo "${str^^[aeiou]}"   # uppercase vowels only
HEllO, WOrld!
```

macOS ships with bash 3.2 by default — these operators will not work there. Use `tr` as a fallback: `echo "$str" | tr '[:upper:]' '[:lower:]'`

Case-insensitive input comparison

Converting input to a known case before comparing is more reliable than trying to match every capitalisation variant.

```
#!/bin/bash
read -r -p "Continue? (yes/no): " answer

if [ "${answer,,}" = "yes" ]; then
    echo "Proceeding..."
else
    echo "Aborted."
fi
# "YES", "Yes", "yes", "yEs" all match cleanly
```

7 — Testing Strings

Before operating on a string it is often useful to check its length, or whether it contains a particular substring. Here are the standard approaches.

Empty, non-empty, and contains checks

```
str="Hello, World!"

# Check if empty (zero length)
if [ -z "$str" ]; then echo "empty"; else echo "not empty"; fi
not empty

# Check if non-empty (non-zero length)
if [ -n "$str" ]; then echo "has content"; fi
has content

# Check if a string contains a substring — use glob matching in [[ ]]
if [[ "$str" == *"World"* ]]; then
    echo "Contains 'World'"
fi
Contains 'World'

# Check if a string starts with a prefix
if [[ "$str" == "Hello"* ]]; then
    echo "Starts with Hello"
fi
Starts with Hello

# Check if a string ends with a suffix
if [[ "$str" == *"!" ]]; then
    echo "Ends with !"
fi
Ends with !

# String equality and inequality
if [ "$str" = "Hello, World!" ]; then echo "equal"; fi
if [ "$str" != "Goodbye"     ]; then echo "not equal"; fi
```

*[[ ]] (double brackets) is needed for glob matching with *. Conditionals are covered fully in Topic 5.*

8 — String Concatenation

Bash has no explicit concatenation operator — you simply place strings and variables next to each other inside double quotes. You can also use += to append to a string variable.

Joining strings

```
# Adjacent values concatenate automatically
first="Hello"
second="World"
combined="$first, $second!"
echo "$combined"
Hello, World!

# += appends to an existing string
msg="Hello"
msg+=", World"
msg+="!"
echo "$msg"
Hello, World!

# Building a string in a loop
csv=""
for item in apple banana cherry; do
    csv+= "$item,"
done
# Strip trailing comma
echo "${csv%,}"
apple, banana, cherry
```

9 — Quick Reference

Arithmetic

SYNTAX	WHAT IT DOES
`$(( a + b ))`	Integer addition (also `-` `*` `/` `%` `**`)
`(( x++ ))`	Post-increment x (side-effect only, no substitution)
`(( x += 5 ))`	Compound assignment (also `-=` `*=` `/=` `%=`)
`let x=a+b`	Alternative arithmetic (legacy — prefer `$(( ))`)
`echo "scale=2; expr" | bc`	Floating-point arithmetic

SYNTAX	WHAT IT DOES
`echo "expr" | bc -l`	Float with maths library (sqrt, sin, cos…)

String Operations

SYNTAX	WHAT IT DOES	EXAMPLE RESULT
`${#var}`	String length	`${#"hello"}` → `5`
`${var:n}`	Substring from index n	`${"hello":2}` → `llo`
`${var:n:len}`	Substring of length len from index n	`${"hello":1:3}` → `ell`
`${var#pat}`	Remove shortest prefix matching pat	`${"file.tar.gz"#*.}` → `tar.gz`
`${var##pat}`	Remove longest prefix matching pat	`${"file.tar.gz"##*.}` → `gz`
`${var%pat}`	Remove shortest suffix matching pat	`${"file.tar.gz"%.*}` → `file.tar`
`${var%%pat}`	Remove longest suffix matching pat	`${"file.tar.gz"%%.*}` → `file`
`${var/pat/rep}`	Replace first occurrence of pat	
`${var//pat/rep}`	Replace all occurrences of pat	
`${var/#pat/rep}`	Replace prefix pat	
`${var/%pat/rep}`	Replace suffix pat	
`${var^^}`	Convert to UPPERCASE (bash 4+)	
`${var,,}`	Convert to lowercase (bash 4+)	
`${var^}`	Capitalise first character (bash 4+)	

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `calc.sh` that accepts two numbers and an operator (`+`, `-`, `*`, `/`) as command-line arguments and prints the result. Division should show two decimal places. If the operator is not one of the four supported ones, print an error to stderr and exit with code 1.

Hint: use a `case` statement on `$2` (the operator). For division use `bc` with `scale=2`; for the others use `$(( ))`. Quote the operator argument carefully to avoid shell interpretation of `*`.

▶ Show Sample Solution

EXERCISE 2

Write a script called `pathinfo.sh` that accepts a full file path as a command-line argument and prints: the directory, the filename, the file extension, and the filename without its extension — all using parameter expansion only (no `basename`, `dirname`, or `cut`).

Hint: use `${path%/*}` for directory, `${path##*/}` for filename, `${filename##*.}` for extension, and `${filename%.*}` for base name.

▶ Show Sample Solution

EXERCISE 3

Write a script called `invoice.sh` that stores a list of at least four item prices as variables, adds them together using arithmetic expansion, calculates 20% VAT, and prints a formatted invoice using `printf` showing each item, the subtotal, the VAT amount, and the grand total — all to 2 decimal places.

Hint: use `bc` with `scale=2` for the float additions and VAT calculation. Use `printf "%-20s £%8.2f\n"` to align the rows.

▶ Show Sample Solution

EXERCISE 4

Write a script called `slugify.sh` that accepts a string argument (a blog post title, for example) and converts it into a URL-friendly slug: lowercase, spaces replaced with hyphens, and any characters that are not letters, numbers, or hyphens removed. For example, `"Hello World! This is Bash 4"` should become `hello-world-this-is-bash-4`.

Hint: chain three operations — use `${title,,}` for lowercase, `${result// /-}` to replace spaces, then pipe through `tr -cd 'a-z0-9-'` to strip non-slug characters.

▶ Show Sample Solution

CHAPTER 5 OF 12

Topic 5 — Conditional Statements

🔀 Topic 5 — Conditional Statements

Conditionals let a script make decisions — running different code depending on whether a condition is true or false. This chapter covers `if/elif/else`, the two test syntaxes `[ ]` and `[[ ]]` with their full set of operators, file tests, logical connectives, the `case` statement for multi-branch matching, and the compact `&& / ||` shorthand. By the end you will be able to write scripts that respond intelligently to their environment and input.

1 — How if Works

In bash, `if` does not test a boolean value — it runs a **command** and checks its **exit status**. If the exit status is `0` (success), the condition is true; any non-zero exit status is false. The test commands `[` and `[[` are simply commands that return 0 or non-zero based on a comparison.

if / elif / else structure

```
#!/bin/bash

score=72

if [ "$score" -ge 90 ]; then
    echo "Grade: A"
elif [ "$score" -ge 70 ]; then
    echo "Grade: B"
elif [ "$score" -ge 50 ]; then
    echo "Grade: C"
else
    echo "Grade: F"
fi
Grade: B
```

The semicolon before `then` is required when `then` is on the same line as `if`. Alternatively, put `then` on the next line and omit the semicolon.

if tests any command. You can use `if grep -q "error" logfile; then` — if grep finds a match (exit 0), the block runs. This means every command in bash is potentially a condition. The `[` and `[[` commands are just the most common ones used with `if`.

2 — [ ] vs [[ ]] — Which to Use

[is a traditional POSIX command (also called `test`) — it is available in every shell. [[is a bash built-in keyword that extends [with extra features and fewer surprises. For bash scripts, prefer [[.

[ ] — POSIX TEST (COMPATIBLE)

```
# Works in any sh-compatible shell
# Variables MUST be quoted
[ "$name" = "Philip" ]

# Logical AND uses -a
[ "$a" -gt 0 -a "$a" -lt 10 ]

# No regex or glob support
# No &&, || inside brackets
```

[[ ]] — BASH KEYWORD (RECOMMENDED)

```
# Bash only — not POSIX sh
# Unquoted variables are safe
[[ $name == "Philip" ]]

# Logical AND uses &&
[[ $a -gt 0 && $a -lt 10 ]]

# Glob matching with ==
[[ $file == *.txt ]]

# Regex matching with =~
[[ $input =~ ^[0-9]+$ ]]
```

⚠ **Always quote variables in [ ].** If `$var` is empty or contains spaces, an unquoted `[ $var = "x" ]` will either throw a syntax error or give wrong results. In `[[ ]]`, word-splitting does not apply so unquoted variables are safe — though quoting is still good practice.

3 — Numeric Comparisons

For comparing integers, bash uses flag-based operators (not the < / > symbols, which mean redirection in this context). These work identically inside both `[ ]` and `[[ ]]`.

OPERATOR	MEANING	EXAMPLE
`-eq`	Equal to	`[ "$a" -eq "$b" ]`
`-ne`	Not equal to	`[ "$a" -ne 0 ]`
`-lt`	Less than	`[ "$a" -lt 10 ]`
`-le`	Less than or equal to	`[ "$a" -le 10 ]`
`-gt`	Greater than	`[ "$a" -gt 0 ]`
`-ge`	Greater than or equal to	`[ "$a" -ge 1 ]`

Numeric comparison in practice

```
age=17

if [[ $age -lt 18 ]]; then
    echo "You must be 18 or older."
elif [[ $age -ge 18 && $age -lt 65 ]]; then
    echo "Standard admission."
else
    echo "Senior discount applies."
fi
You must be 18 or older.

# You can also use (( )) for numeric conditions — reads more naturally
if (( age >= 18 && age < 65 )); then
    echo "Standard admission."
fi
```

(( )) uses C-style comparison symbols (> < == !=) and does not need $ on variable names. It is often the most readable choice for pure numeric tests.

4 — String Comparisons

OPERATOR	MEANING	NOTES
`=` or `==`	Strings are equal	*Use `=` in `[ ]`, either in `[[ ]]`. In `[[ ]]`, the right side is treated as a glob pattern.*
`!=`	Strings are not equal	
`<`	Lexicographically less than	*In `[ ]`, must escape: `\<`. In `[[ ]]`, use as-is.*
`>`	Lexicographically greater than	*Same escaping caveat as `<`.*
`-z`	String is empty (zero length)	`[ -z "$var" ]`
`-n`	String is non-empty	`[ -n "$var" ]`
`=~`	String matches a regex	`[[ ]]` *only. Do not quote the pattern.*

String comparison examples

```
name="Philip"

# Equality
if [[ "$name" == "Philip" ]]; then echo "Hello, Philip!"; fi
Hello, Philip!

# Glob matching — right side is a pattern, not quoted
if [[ "$name" == Ph* ]]; then echo "Starts with Ph"; fi
Starts with Ph

# Regex matching with =~ (POSIX extended regex)
email="user@example.com"
if [[ "$email" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email format"
fi
Valid email format

# Empty / non-empty checks
input=""
if [[ -z "$input" ]]; then echo "Input is empty"; fi
Input is empty

# Lexicographic ordering
if [[ "apple" < "banana" ]]; then echo "apple comes first"; fi
apple comes first
```

When using =~, capture groups are stored in the BASH_REMATCH array: ${BASH_REMATCH[0]} is the full match, ${BASH_REMATCH[1]} is the first group, etc.

5 — File Test Operators

File tests check properties of files and directories — whether they exist, what type they are, and what permissions they have. These are some of the most frequently used tests in real-world scripts.

OPERATOR	TRUE IF...
`-e file`	File exists (any type)
`-f file`	File exists and is a regular file
`-d file`	File exists and is a directory

OPERATOR	TRUE IF...
`-L file`	File exists and is a symbolic link
`-r file`	File exists and is readable by the current user
`-w file`	File exists and is writable by the current user
`-x file`	File exists and is executable by the current user
`-s file`	File exists and has a size greater than zero
`-z file`	File exists and has a size of zero
`-b file`	File is a block device
`-c file`	File is a character device
`-p file`	File is a named pipe (FIFO)
`f1 -nt f2`	f1 is newer than f2 (modification time)
`f1 -ot f2`	f1 is older than f2
`f1 -ef f2`	f1 and f2 refer to the same file (hard link or same inode)

File tests in a script

```
#!/bin/bash
path="$1"

if [[ -z "$path" ]]; then
    echo "Usage: $0 <path>" >&2
    exit 1
fi

if [[ ! -e "$path" ]]; then
    echo "'$path' does not exist."
elif [[ -d "$path" ]]; then
    echo "'$path' is a directory."
elif [[ -f "$path" ]]; then
    if [[ -r "$path" && -w "$path" ]]; then
        echo "'$path' is a readable and writable file."
    elif [[ -r "$path" ]]; then
        echo "'$path' is readable but not writable."
    else
        echo "'$path' exists but is not readable."
    fi
else
    echo "'$path' exists but is not a regular file or directory."
fi
```

6 — Logical Operators

Logical operators let you combine multiple conditions into a single test. The syntax differs slightly between `[ ]` and `[[ ]]`.

OPERATOR	IN [ ]	IN [[ ]]	MEANING
AND	`-a`	`&&`	Both conditions must be true
OR	`-o`	`||`	At least one condition must be true
NOT	`!`	`!`	Negate the condition

Combining conditions

```
age=25
member="yes"

# AND — both must be true
if [[ $age -ge 18 && "$member" == "yes" ]]; then
    echo "Access granted."
fi
Access granted.

# OR — either condition is enough
role="admin"
if [[ "$role" == "admin" || "$role" == "superuser" ]]; then
    echo "Elevated privileges."
fi
Elevated privileges.

# NOT — negate a condition
file="config.cfg"
if [[ ! -f "$file" ]]; then
    echo "Config file missing — creating default."
    touch "$file"
fi

# Combining three or more conditions
if [[ $age -ge 18 && $age -lt 65 && "$member" == "yes" ]]; then
    echo "Full member benefits apply."
fi
```

Chaining with && and || Outside Brackets

The `&&` and `||` operators can also be used *outside* brackets to chain commands — running the second command only if the first succeeded or failed.

Short-circuit command chaining

```
# cmd1 && cmd2 — run cmd2 only if cmd1 succeeds (exit 0)
mkdir -p /tmp/mydir && echo "Directory created."

# cmd1 || cmd2 — run cmd2 only if cmd1 FAILS (non-zero exit)
cd /nonexistent || echo "ERROR: directory not found." >&2

# Common pattern: exit on failure
cp source.txt dest.txt || { echo "Copy failed" >&2; exit 1; }

# Guard clause — ensure a directory exists before writing
[[ -d "$output_dir" ]] || mkdir -p "$output_dir"
```

Note the `{ }` grouping in the third example — without the braces, only `echo` would be the "or" branch; `exit 1` would always run. Braces group multiple commands into one for `||`.

7 — The case Statement

When you need to match a value against many possible patterns, a `case` statement is far cleaner than a long chain of `elif` blocks. Each branch uses glob-style patterns and ends with `;;`.

case syntax

```
#!/bin/bash

case "$1" in
    start)
        echo "Starting the service..."
        ;;
    stop)
        echo "Stopping the service..."
        ;;
    restart)
        echo "Restarting the service..."
        ;;
    status)
        echo "Checking status..."
        ;;
    *)
        echo "Usage: $0 {start|stop|restart|status}" >&2
        exit 1
        ;;
esac
```

Multiple Patterns per Branch

Separate patterns with | to match several values in one branch.

Matching multiple values and using globs

```
#!/bin/bash
read -r -p "Enter a file name: " fname

case "${fname,,}" in           # ${fname,,} lowercases input first
    *.jpg | *.jpeg | *.png | *.gif | *.webp)
        echo "Image file detected."
        ;;
    *.mp4 | *.mkv | *.avi | *.mov)
        echo "Video file detected."
        ;;
    *.sh | *.bash)
        echo "Shell script detected."
        ;;
    *.txt | *.md | *.csv)
        echo "Text file detected."
        ;;
    "")
        echo "No filename entered."
        ;;
    *)
        echo "Unknown file type."
        ;;
esac
```

Fall-through with ;& and ;;&

Bash 4+ fall-through syntax

```
level="gold"

case "$level" in
    platinum)
        echo "Platinum perk: lounge access."
        ;&              # ;& falls through to the NEXT branch unconditionally
    gold)
        echo "Gold perk: priority boarding."
        ;&
    silver)
        echo "Silver perk: extra baggage."
        ;;
    *)
        echo "Standard tier."
        ;;
esac
Gold perk: priority boarding.
Silver perk: extra baggage.
# ;; stops. ;& continues to next. ;;& re-tests remaining patterns.
```

Fall-through with `;&` is bash 4+ only and is rarely needed. The more common `;;` is the standard terminator — it stops after the matching branch.

8 — Practical Patterns

Validate a numeric argument

Checking that an argument is a positive integer

```
#!/bin/bash
input="$1"

if [[ ! "$input" =~ ^[0-9]+$ ]]; then
    echo "ERROR: '$input' is not a positive integer." >&2
    exit 1
fi
echo "Valid number: $input"
```

Require a file to exist before proceeding

Guard clause pattern — fail early

```
#!/bin/bash
config="$HOME/.myapp/config"

[[ -f "$config" ]] || { echo "Config not found: $config" >&2; exit 1; }
[[ -r "$config" ]] || { echo "Config not readable." >&2; exit 1; }

# Only reaches here if both tests passed
echo "Loading config from $config..."
```

Interactive yes/no prompt

A reusable confirm() function

```
#!/bin/bash
confirm() {
    read -r -p "$1 [y/N]: " response
    case "${response,,}" in
        y | yes) return 0 ;;    # return 0 = true
        *)       return 1 ;;    # return 1 = false
    esac
}

if confirm "Delete all log files?"; then
    rm -f /var/log/myapp/*.log
    echo "Logs deleted."
else
    echo "Cancelled."
fi
```

9 — Quick Reference

SYNTAX	WHAT IT DOES
`if cmd; then … fi`	Runs if cmd exits with 0
`if [ expr ]; then … fi`	POSIX test — quote all variables
`if [[ expr ]]; then … fi`	Bash test — glob + regex, safer with variables

SYNTAX	WHAT IT DOES
`if (( expr )); then … fi`	Arithmetic test — C-style operators
`-eq -ne -lt -le -gt -ge`	Numeric comparisons (inside `[ ]` or `[[ ]]`)
`= != < > -z -n`	String comparisons
`=~`	Regex match (`[[ ]]` only)
`-e -f -d -r -w -x -s -L`	File tests
`&& || !`	Logical AND, OR, NOT inside `[[ ]]`
`-a -o !`	Logical AND, OR, NOT inside `[ ]`
`cmd1 && cmd2`	Run cmd2 only if cmd1 succeeds
`cmd1 || cmd2`	Run cmd2 only if cmd1 fails
`case "$var" in pat) … ;; esac`	Multi-branch pattern matching
`pat1 | pat2)`	Match either pattern in a case branch
`${BASH_REMATCH[n]}`	Regex capture groups from `=~` match

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `filecheck.sh` that accepts a file path as an argument and reports: whether the path exists; if it does, whether it is a file or directory; and if it is a file, whether it is readable, writable, and/or executable. If no argument is given, print a usage message to stderr and exit with code 1.

Hint: use nested `if` blocks with `-e`, `-f`, `-d`, `-r`, `-w`, `-x`. Guard against missing input with `[[ -z "$1" ]]`.

▶ Show Sample Solution

EXERCISE 2

Write a script called `validate.sh` that prompts the user for an email address and a port number, validates both using =~ regex, and prints a clear pass/fail result for each. A valid port is a number between 1 and 65535.

Hint: use `[[ "$email" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]` for email. For port, first check it is all digits with `=~ ^[0-9]+$`, then use `(( port >= 1 && port <= 65535 ))` for range.

▶ Show Sample Solution

EXERCISE 3

Write a script called `daytype.sh` that accepts a day name as an argument (e.g. `Monday`) and uses a `case` statement to print whether it is a weekday, weekend, or an unrecognised input. The check should be case-insensitive, so `saturday`, `Saturday`, and `SATURDAY` all give the same result.

Hint: use `${1,,}` to convert the argument to lowercase before the `case` statement, then match against the lowercase day names. Use `|` to group Monday–Friday in one branch.

▶ Show Sample Solution

EXERCISE 4

Write a script called `backup_check.sh` that checks whether a backup directory (passed as an argument) exists and is writable, whether the directory contains any `.tar.gz` files, and whether the most recently modified `.tar.gz` file is newer than a file called `last_backup.txt` in the same directory. Print a clear status report for each check.

Hint: use `-d` and `-w` for the directory; use `ls *.tar.gz 2>/dev/null` with `$?` to check for archives; use `-nt` to compare modification times between files.

▶ Show Sample Solution

CHAPTER 6 OF 12

Topic 6 — Loops

🔁 Topic 6 — Loops

Loops let a script repeat a block of commands — iterating over a list, counting through a range, reading every line of a file, or running until a condition changes. Bash provides four loop constructs, each suited to a different situation. This chapter covers all four, along with `break`, `continue`, the IFS-aware file-reading pattern, loop output redirection, and the interactive `select` menu loop.

1 — Which Loop to Use

LIST ITERATION

`for item in list`

Iterate over a fixed list of words, a glob, or the output of a command. Best choice when you know the items in advance.

COUNTED ITERATION

`for (( i=0; i<N; i++ ))`

C-style numeric loop. Best when you need an index counter or a precise range with a step.

CONDITION AT TOP

`while condition`

Runs while the condition is true, checking *before* each iteration. May run zero times. Also the standard pattern for reading a file line by line.

CONDITION AT TOP (INVERTED)

`until condition`

Runs *until* the condition becomes true — the opposite of while. Useful for "keep trying until it works" patterns.

2 — for Loop: List Style

The list-style `for` loop assigns each word in a list to a variable in turn and runs the loop body for each one. The list can be a literal sequence, a brace expansion, a glob pattern, or the output of a command.

Iterating over lists

```
#!/bin/bash

# Literal list
for fruit in apple banana cherry mango; do
    echo "Fruit: $fruit"
done
Fruit: apple
Fruit: banana
Fruit: cherry
Fruit: mango

# Brace expansion — {start..end} or {start..end..step}
for i in {1..5}; do
    echo -n "$i "
done
echo
1 2 3 4 5

for i in {0..20..5}; do   # step of 5
    echo -n "$i "
done
echo
0 5 10 15 20

# Glob — iterate over matching files
for file in /etc/*.conf; do
    echo "Config: $file"
done

# Command substitution — iterate over output lines
for user in $(cut -d: -f1 /etc/passwd | head -5); do
    echo "User: $user"
done
```

When iterating over command output with `$( )`, word splitting applies — lines with spaces are split into multiple items. Use a `while read` loop (section 5) when lines may contain spaces.

Looping Over Script Arguments

Processing all arguments passed to a script

```
#!/bin/bash
# Process every argument passed to this script

for arg in "$@"; do
    echo "Processing: $arg"
done

# Shorthand — omitting 'in "$@"' is equivalent
for arg; do
    echo "Processing: $arg"
done
```

Always use "$@" (not $) to preserve arguments that contain spaces — each argument stays as a single item regardless of internal whitespace.*

3 — for Loop: C Style

The C-style `for` loop uses arithmetic expressions for initialisation, condition, and update. It is the best choice when you need a numeric index or a loop that counts with a custom step.

C-style for loop

```
# Basic: count from 1 to 10
for (( i=1; i<=10; i++ )); do
    echo -n "$i "
done
echo
1 2 3 4 5 6 7 8 9 10

# Count down
for (( i=5; i>0; i-- )); do
    echo -n "$i "
done
echo
5 4 3 2 1

# Step by 2
for (( i=0; i<=10; i+=2 )); do
    echo -n "$i "
done
echo
0 2 4 6 8 10

# Using the index to access a positional parameter
for (( i=1; i<=$#; i++ )); do
    echo "Arg $i: ${!i}"   # ${!i} = indirect expansion — value of the i-th arg
done
```

> **Brace expansion vs C-style for ranges:** Use `{1..10}` when the range limits are fixed literals. Use `(( i=1; i<=n; i++ ))` when the end value is a variable — brace expansion does not expand variables: `{1..$n}` does *not* work.

4 — while Loop

A `while` loop evaluates its condition *before* each iteration and runs the body as long as the condition is true (exit status 0). It is the right choice when you don't know how many iterations are needed in advance.

while loop patterns

```
#!/bin/bash

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

# Infinite loop — runs until broken from inside
while true; do
    read -r -p "Enter 'quit' to exit: " input
    if [[ "$input" == "quit" ]]; then
        echo "Goodbye!"
        break
    fi
    echo "You typed: $input"
done

# Retry with a limit — keep trying until success or max attempts
attempts=0
max=3
while (( attempts < max )); do
    read -r -s -p "Password: " pw; echo
    if [[ "$pw" == "secret" ]]; then
        echo "Access granted."; break
    fi
    (( attempts++ ))
    echo "Wrong. $((max - attempts)) attempt(s) remaining."
done
[[ $attempts -ge $max ]] && echo "Locked out."
```

5 — Reading a File Line by Line

The most important and most common use of `while` in real scripts is reading a file line by line. The canonical pattern is:

```
while IFS= read -r line; do
    # process $line
done < "$filename"
```

Each part of that one-liner matters:

- **`IFS=`** — sets the Internal Field Separator to empty for this one command, preventing `read` from stripping leading and trailing whitespace from each line.
- **`read -r`** — raw mode: backslashes are treated literally, not as escape characters.
- **`< "$filename"`** — redirects the file into the loop's stdin. The redirection goes on the `done` line, not on the `while` line.

Line-by-line file reading patterns

```
#!/bin/bash

# Basic: print each line with a line number
linenum=0
while IFS= read -r line; do
    (( linenum++ ))
    printf "%4d  %s\n" "$linenum" "$line"
done < /etc/hosts

# Skip blank lines and comment lines
while IFS= read -r line; do
    [[ -z "$line" ]]           && continue   # skip blank
    [[ "$line" == \#* ]]        && continue   # skip comments
    echo "Active entry: $line"
done < /etc/hosts

# Split each line into fields using IFS
# /etc/passwd is colon-delimited: user:x:uid:gid:comment:home:shell
while IFS=':' read -r user _pw uid gid comment home shell; do
    printf "%-15s  uid=%-6s  %s\n" "$user" "$uid" "$shell"
done < /etc/passwd

# Read from a pipe (note: variables set inside may not persist)
ps aux | while IFS= read -r line; do
    [[ "$line" == *"bash"* ]] && echo "$line"
done
```

When you pipe into a while loop (`cmd | while … done`), the loop body runs in a subshell on most systems — variables set inside will not be visible after the loop. Use `while … done < <(cmd)` (process substitution) to avoid this.

6 — until Loop

`until` is the logical inverse of `while`: it runs the loop body as long as the condition is *false*, stopping when the condition becomes true. Every `until` loop can be rewritten as a `while` with a negated condition — use whichever reads more naturally.

until loop examples

```
# Wait until a file appears
echo "Waiting for /tmp/ready.flag ..."
until [[ -f "/tmp/ready.flag" ]]; do
    sleep 1
done
echo "Flag found — proceeding."

# Equivalent with while (negated condition):
while [[ ! -f "/tmp/ready.flag" ]]; do
    sleep 1
done

# Count up with until
n=1
until (( n > 5 )); do
    echo "n = $n"
    (( n++ ))
done
```

7 — break and continue

`break` exits the enclosing loop immediately. `continue` skips the rest of the current iteration and moves to the next one. Both accept an optional numeric argument to target an outer loop when loops are nested.

break and continue

```
# continue — skip even numbers, print only odds 1-10
for (( i=1; i<=10; i++ )); do
    (( i % 2 == 0 )) && continue
    echo -n "$i "
done
echo
1 3 5 7 9

# break — stop at the first file larger than 1 MB
for file in /var/log/*.log; do
    size=$(stat -c%s "$file")
    if (( size > 1048576 )); then
        echo "Large file found: $file ($size bytes)"
        break
    fi
done

# Nested loops — break 2 exits both the inner AND outer loop
for i in {1..3}; do
    for j in {1..3}; do
        if (( i == 2 && j == 2 )); then
            echo "Breaking out of both loops at i=$i j=$j"
            break 2
        fi
        echo "i=$i j=$j"
    done
done
i=1 j=1
i=1 j=2
i=1 j=3
i=2 j=1
Breaking out of both loops at i=2 j=2
```

8 — Redirecting Loop Output

You can redirect the entire output of a loop — or pipe it — by placing the redirection operator after `done`. This is far cleaner than redirecting inside every `echo` call.

Redirecting and piping loop output

```
# Write the entire loop's output to a file
for i in {1..5}; do
    echo "Line $i"
done > output.txt

# Append loop output to a log file
for file in *.sh; do
    echo "$(date): processing $file"
done >> process.log

# Pipe a loop's output to another command
for name in charlie alice bob diana; do
    echo "$name"
done | sort
alice
bob
charlie
diana

# Capture loop output into a variable
result=$(
    for i in {1..3}; do
        echo "item$i"
    done
)
echo "$result"
```

9 — select: Interactive Menus

`select` is a special loop that displays a numbered menu from a list and prompts the user to choose an option. It is the standard way to build an interactive menu in a bash script.

Building a menu with select

```
#!/bin/bash

# PS3 is the prompt shown to the user (default is "#? ")
PS3="Choose an action: "

select choice in "Show disk usage" "Show uptime" "List users" "Quit"; do
    case "$choice" in
        "Show disk usage")  df -h / ;;
        "Show uptime")      uptime ;;
        "List users")       cut -d: -f1 /etc/passwd ;;
        "Quit")             echo "Bye!"; break ;;
        *)                  echo "Invalid choice: $REPLY" ;;
    esac
done
```

`$REPLY` holds the raw text the user typed. `$choice` holds the corresponding menu item string (or empty if the number was out of range). The menu is redisplayed after each selection unless you `break`.

10 — IFS and Word Splitting in Loops

The **Internal Field Separator** (`IFS`) controls how bash splits words. Its default value is space, tab, and newline. Changing `IFS` in a loop lets you split on any delimiter — very useful for processing CSV or colon-separated data.

Changing IFS to parse delimited data

```
# Split a CSV string into fields
record="Philip,32,London,Engineer"

IFS=',' read -r -a fields <<< "$record"
echo "Name : ${fields[0]}"
echo "Age  : ${fields[1]}"
echo "City : ${fields[2]}"

# Loop through a CSV file, one record per line
while IFS=',' read -r name age city; do
    printf "%-15s %-5s %s\n" "$name" "$age" "$city"
done < people.csv

# Temporarily change IFS for a for loop — restore afterwards
old_IFS="$IFS"
IFS=':'
for dir in $PATH; do      # $PATH split on : without quotes
    echo "$dir"
done
IFS="$old_IFS"
```

⚠ **Always restore IFS.** If you change IFS globally (not with the `IFS= read` single-command form), save the original first and restore it afterwards. A changed IFS will silently break other parts of your script that rely on word splitting.

11 — Quick Reference

SYNTAX	WHAT IT DOES
`for x in list; do … done`	Iterate over a list of words
`for x in "$@"; do … done`	Iterate over all script arguments
`for x in *.txt; do … done`	Iterate over matching files (glob)
`for x in {1..10}; do … done`	Iterate over a brace-expanded range
`for x in {0..20..5}; do … done`	Range with step
`for (( i=0; i<N; i++ )); do … done`	C-style counted loop

SYNTAX	WHAT IT DOES
`while condition; do … done`	Run while condition is true
`while true; do … done`	Infinite loop (exit with break)
`until condition; do … done`	Run until condition becomes true
`while IFS= read -r line; do … done < file`	Read a file line by line
`while IFS=',' read -r a b c; do … done < file`	Read and split delimited file
`break`	Exit the enclosing loop
`break N`	Exit N levels of nested loops
`continue`	Skip to next iteration
`done > file`	Redirect entire loop output to file
`done | cmd`	Pipe entire loop output to a command
`select x in list; do … done`	Display numbered menu, prompt for choice
`$REPLY`	Raw input from select prompt

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `rename_ext.sh` that accepts two arguments — an old extension and a new extension (e.g. `./rename_ext.sh txt md`) — and renames all files in the current directory with the old extension to use the new one. Print a line for each file renamed, or a message if no matching files are found. Run it in a test directory with dummy files.

Hint: use `for file in *."$1"` to match files with the given extension. Use `${file%.*}` to strip the extension, then append the new one. Use `mv "$file" "$newname"` to rename.

▶ Show Sample Solution

EXERCISE 2

Write a script called `csv_report.sh` that reads a CSV file (create a sample one first with columns `name,score,grade`) line by line, skips the header row, and prints a formatted table. Count how many students passed (grade A or B) and print the total at the end.

Hint: use `while IFS=',' read -r name score grade` with input redirected from the CSV. Use a counter variable and a `case` or `[[ ]]` test on `$grade` to count passes. Skip the header by using a flag variable or `read` once before the loop.

▶ Show Sample Solution

EXERCISE 3

Write a script called `times_table.sh` that accepts a number as an argument and prints its times table from 1 to 12. Then extend it: if no argument is given, use a `select` menu to let the user choose a number from 2 to 12, then print that table.

Hint: use a C-style `for (( i=1; i<=12; i++ ))` loop with `printf` for alignment. For the `select` menu, build the list with brace expansion: `select n in {2..12}`.

▶ Show Sample Solution

EXERCISE 4

Write a script called `disk_watch.sh` that uses a `while` loop to check disk usage on `/` every 3 seconds. Each iteration it should print the current usage percentage and the time. If usage exceeds 80%, print a warning and exit. After 5 checks with no alert, print "All clear" and exit normally.

Hint: use `df / | tail -1 | awk '{print $5}'` to get the usage percentage (it returns something like 42%). Strip the % with `${pct%\%}` before comparing numerically. Use a counter to track iterations.

▶ Show Sample Solution

CHAPTER 7 OF 12

Topic 7 — Functions

🧩 Topic 7 — Functions

Functions let you group commands into a named, reusable block. Instead of copying the same ten lines in three places, you write them once as a function and call it wherever needed. This chapter covers both ways to define a function, how arguments and return values work, the critical importance of `local` variables, recursive functions, and how to split your code across multiple files using `source`.

1 — Defining and Calling Functions

There are two syntactically equivalent ways to define a function in bash. Both are in common use — pick one and be consistent.

STYLE 1 — FUNCTION KEYWORD

```
function greet() {
    echo "Hello, World!"
}

# Call it — just use the name
greet
```

STYLE 2 — POSIX (NO KEYWORD)

```
greet() {
    echo "Hello, World!"
}

# Call it — same way
greet
```

Style 2 (no keyword) is preferred for portability — it works in bash, zsh, ksh, and any POSIX-compliant shell. Style 1 with the `function` keyword is bash-specific but makes functions visually obvious when scanning a file. In bash-only scripts, either is fine.

Functions must be defined before they are called

```
#!/bin/bash

# ✓ Define first, then call
say_hello() {
    echo "Hello!"
}
say_hello

# ✗ Calling before defining — bash error: command not found
say_goodbye                         # error here
say_goodbye() { echo "Goodbye!"; }

# Common pattern: define all functions at the top,
# then put the main logic at the bottom.
main() {
    say_hello
}
main
```

A popular convention is to wrap the main script logic in a `main()` function and call it at the very end. This lets you define all helper functions above `main()` without worrying about order.

2 — Function Arguments

Inside a function, `$1`, `$2`, `$@`, `$#`, and `$*` refer to the *function's own arguments*, not the script's arguments. This is the same set of positional parameter variables — they are just scoped to the function call.

Passing and accessing arguments

```
#!/bin/bash

greet() {
    local name="$1"
    local title="${2:-Mr/Ms}"       # default value if $2 not given
    echo "Hello, $title $name!"
}

greet "Philip" "Dr"
Hello, Dr Philip!

greet "Philip"
Hello, Mr/Ms Philip!

_____________________________________________________
print_all() {
    echo "Number of args : $#"
    echo "All args       : $@"
    for arg in "$@"; do
        echo "  - $arg"
    done
}

print_all apple "banana split" cherry
Number of args : 3
All args       : apple banana split cherry
  - apple
  - banana split
  - cherry
```

The script's own positional parameters (`$1`, `$2`, etc.) are still accessible inside a function — they are just shadowed by the function's arguments. To access the original script arguments from within a function, save them to variables before calling the function.

3 — Local Variables and Scope

By default, every variable in bash is **global** — a variable set inside a function is visible everywhere in the script, and can accidentally overwrite a variable with the same name in the calling code. Use `local` to declare a variable that exists only within the function.

The danger of globals — and how local fixes it

```
#!/bin/bash

# ── Without local — BAD ──────────────────────────
double_bad() {
    result=$(( $1 * 2 ))      # sets the GLOBAL variable 'result'
}
result="original"
double_bad 5
echo "$result"
10                            # 'original' was silently overwritten!

# ── With local — GOOD ────────────────────────────
double_good() {
    local result=$(( $1 * 2 ))   # only exists inside this function
    echo "$result"
}
result="original"
double_good 5
10
echo "$result"
original                      # global is untouched
```

⚠ **Always declare local variables with `local`.** This is the single most important function-writing habit in bash. Failing to use `local` is a common source of subtle, hard-to-debug bugs where a helper function silently corrupts a variable in the caller.

local — declaration forms

```
example() {
    local name="Philip"          # local and assigned
    local count                   # local but unset (empty string)
    local x=1 y=2 z=3            # multiple on one line
    local -r MAX=100             # local AND read-only
    local -i total=0             # local integer
    local -a items=()             # local array
    # ...
}
```

4 — Return Values

Bash functions can "return" in two fundamentally different ways — and which you use depends on whether you need an exit status or an actual data value.

Method 1 — return (exit status only)

`return N` sets the function's exit status to N (0–255). Like a command's exit status, 0 means success and non-zero means failure. The caller reads it via `$?`.

Using return for pass/fail results

```
is_even() {
    local n="$1"
    (( n % 2 == 0 ))    # (( )) sets exit status: 0 if true, 1 if false
                        # no explicit return needed — last command's status is used
}

if is_even 4; then
    echo "4 is even"
fi
4 is even

if ! is_even 7; then
    echo "7 is odd"
fi
7 is odd

─────────────────────────────────────────────────
validate_age() {
    local age="$1"
    [[ "$age" =~ ^[0-9]+$ ]] || return 1   # not numeric
    (( age >= 1 && age <= 120 ))              # in valid range
}

validate_age "25"  && echo "Valid"  || echo "Invalid"
Valid
validate_age "abc" && echo "Valid"  || echo "Invalid"
Invalid
```

Method 2 — echo (capture output)

To return an actual string or number, `echo` it from the function and capture it with command substitution. This is the standard way to "return a value" in bash.

Returning data values via echo

```
to_upper() {
    echo "${1^^}"
}

result=$(to_upper "hello world")
echo "$result"
HELLO WORLD

_____________________________________________________

add() {
    echo $(( $1 + $2 ))
}

sum=$(add 15 27)
echo "Sum: $sum"
Sum: 42

_____________________________________________________

# You can use BOTH at the same time:
# echo the data value AND set the exit status
safe_divide() {
    local a="$1" b="$2"
    if (( b == 0 )); then
        echo "ERROR: division by zero" >&2
        return 1
    fi
    echo "scale=4; $a / $b" | bc
}

if val=$(safe_divide 10 3); then
    echo "Result: $val"
else
    echo "Calculation failed."
fi
Result: 3.3333
```

Be careful with the echo-return pattern: any `echo` or `printf` inside the function becomes part of its "return value" when captured. Use `echo "..." >&2` for debug output you do not want captured.

Method 3 — nameref (bash 4.3+)

A nameref variable (`local -n`) is a reference to another variable by name. It lets a function write a result into a caller-supplied variable name — avoiding a subshell entirely.

Returning via a nameref variable

```
repeat_str() {
    local -n _out="$1"    # -n makes _out a reference to the variable named by $1
    local str="$2"
    local n="$3"
    _out=""
    for (( i=0; i<n; i++ )); do
        _out+="$str"
    done
}

repeat_str my_result "ab" 4
echo "$my_result"
abababab
```

Namerefs avoid the performance cost of a subshell — useful in tight loops. Avoid naming the nameref variable the same as the caller's variable (e.g. don't use `-n result` if the caller also has a `result` variable) — it causes a circular reference.

5 — Recursive Functions

A function can call itself — this is called recursion. Each call gets its own local variable scope, so the variables from one level don't interfere with another. Bash supports recursion but has no tail-call optimisation, so deep recursion is slow and risks hitting stack limits. Keep recursive depths shallow.

Factorial and directory tree recursion

```
# Classic recursive factorial
factorial() {
    local n="$1"
    (( n <= 1 )) && { echo 1; return; }
    local prev=$(factorial $(( n - 1 )))
    echo $(( n * prev ))
}

echo "5! = $(factorial 5)"
5! = 120
echo "10! = $(factorial 10)"
10! = 3628800

# ──────────────────────────────────────────────────

# Recursive directory listing with indentation
list_tree() {
    local dir="$1"
    local indent="$2"
    local item

    for item in "$dir"/*; do
        [[ -e "$item" ]] || continue
        echo "${indent}$(basename "$item")"
        [[ -d "$item" ]] && list_tree "$item" "${indent}  "
    done
}

list_tree /etc/ssh ""
```

6 — Function Libraries and source

Once you have a collection of useful functions, you can save them in a separate file and load them into any script using `source` (or its shorthand `.`). This is how shared utility libraries work in bash.

Creating and sourcing a library file

```
# ── lib/utils.sh — the shared library ──────────────────
#!/bin/bash
# Guard against being sourced more than once
[[ -n "${_UTILS_LOADED:-}" ]] && return
readonly _UTILS_LOADED=1

log_info() {
    printf "[INFO]  %s %s\n" "$(date +%H:%M:%S)" "$*"
}

log_warn() {
    printf "[WARN]  %s %s\n" "$(date +%H:%M:%S)" "$*" >&2
}

log_error() {
    printf "[ERROR] %s %s\n" "$(date +%H:%M:%S)" "$*" >&2
}

die() {
    log_error "$1"
    exit "${2:-1}"
}

require_cmd() {
    command -v "$1" >/dev/null 2>&1 || \
        die "Required command not found: $1"
}
```

```
# ── myscript.sh — loads the library ────────────────────
#!/bin/bash

# Get the directory of this script, then source the library
SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)
# shellcheck source=lib/utils.sh
source "$SCRIPT_DIR/lib/utils.sh"

require_cmd curl
log_info "Starting backup..."
log_warn "Disk space is low."
log_info "Done."
[INFO]  10:32:45 Starting backup...
[WARN]  10:32:45 Disk space is low.
[INFO]  10:32:45 Done.
```

`${BASH_SOURCE[0]}` gives the path to the current file even when it is sourced — unlike `$0` which gives the top-level script's name. Always use `BASH_SOURCE` in library files.

source vs . — `source file` and `. file` are identical. Both run the file in the *current shell* (not a subshell), so any functions and variables defined in it become available immediately. The dot form is POSIX-standard; `source` is bash-specific but more readable.

7 — Advanced Patterns

Functions that validate arguments

Guard clauses inside a function

Checking arguments at the top of a function and returning early keeps the main logic uncluttered.

```
create_backup() {
    local src="$1"
    local dest="$2"

    [[ -n "$src"  ]] || { log_error "src required";  return 1; }
    [[ -n "$dest" ]] || { log_error "dest required"; return 1; }
    [[ -e "$src"  ]] || { log_error "src does not exist: $src"; return 1; }
    [[ -d "$dest" ]] || mkdir -p "$dest"

    cp -r "$src" "$dest/" && log_info "Backed up $src to $dest"
}
```

Storing and passing functions

Using functions as callbacks via export -f

A function can be exported so it is available in child processes — useful when using `xargs` or `parallel`.

```
process_file() {
    echo "Processing: $1 (size: $(wc -c < "$1") bytes)"
}

# Export the function so subshells can see it
export -f process_file

# Run it via xargs — each file is processed in a subshell
find /tmp -name "*.log" | xargs -I{} bash -c 'process_file "$@"' _ {}
```

Using command to call a function safely

Checking whether a function exists

```
# Check if a function is defined before calling it
if declare -f my_function >/dev/null; then
    my_function
else
    echo "my_function is not defined"
fi

# List all defined functions
declare -F              # prints: declare -f function_name for each
declare -F | awk '{print $3}'    # just the names
```

8 — Quick Reference

SYNTAX	WHAT IT DOES
`name() { … }`	Define a function (POSIX style)
`function name() { … }`	Define a function (bash style)
`name arg1 arg2`	Call a function with arguments
`$1 $2 … $# $@`	Function's own positional parameters
`local var=value`	Declare a variable local to the function
`local -r var=value`	Local read-only variable

SYNTAX	WHAT IT DOES
`local -i var=0`	Local integer variable
`local -a arr=()`	Local array variable
`local -n ref="$1"`	Nameref — reference to caller's variable (bash 4.3+)
`return N`	Exit function with status N (0 = success)
`result=$(fn arg)`	Capture function's printed output as a value
`source file` or `. file`	Load and execute a file in the current shell
`export -f name`	Export function to child processes
`declare -f name`	Check if a function is defined (exit 0 if yes)
`declare -F`	List all currently defined function names
`${BASH_SOURCE[0]}`	Path to the current file (works even when sourced)

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Create a shared library file `lib/log.sh` that defines four logging functions: `log_info`, `log_warn`, `log_error`, and `log_debug`. Each should print a timestamp, a level label, and the message. Then write a script `app.sh` that sources the library and calls each function. Add a global variable `LOG_LEVEL` (default `INFO`) that suppresses `log_debug` output unless `LOG_LEVEL=DEBUG` is set.

Hint: use `printf "[%-5s] %s %s\n"` for consistent label width. In `log_debug`, check `[[ "${LOG_LEVEL:-INFO}" == "DEBUG" ]]` before printing. Source with `source "$(dirname "$0")/lib/log.sh"`.

▶ Show Sample Solution

EXERCISE 2

Write a script called `string_utils.sh` that defines three functions: `str_repeat` (repeat a string N times), `str_pad` (pad a string to a given width with a pad character), and `str_trim` (remove leading and trailing whitespace). Each function should print its result so it can be captured with `$( )`. Include test calls at the bottom demonstrating each function.

Hint: for `str_repeat` use a C-style for loop appending to a local variable. For `str_pad` use `printf "%-Ns"` with a calculated width. For `str_trim` use parameter expansion: `${var#"${var%%[![:space:]]*}"}` strips the leading spaces.

▶ Show Sample Solution

EXERCISE 3

Write a script called `fibonacci.sh` that uses a recursive function to calculate the Nth Fibonacci number. Then add a second, iterative version of the same function and compare their outputs. Call both with the same input (try N=10 and N=15) and print the results side by side.

Hint: the recursive version is the classic `fib(n) = fib(n-1) + fib(n-2)` with base cases 0 and 1. For the iterative version, use a while loop with two tracking variables `a` and `b`, swapping values each iteration.

▶ Show Sample Solution

EXERCISE 4

Build a small script called `menu_app.sh` that uses functions to structure a multi-option interactive application. Define separate functions for at least three actions (e.g. `show_system_info`, `show_disk_usage`, `show_top_processes`), a `show_menu` function using `select`, and a `main` function that calls `show_menu` in a loop. The menu should include a "Quit" option that exits cleanly.

Hint: call each action function from the case branch inside `show_menu`. In `main`, use `while true; do show_menu; done`. Have the "Quit" branch call `exit 0` or set a flag variable that causes `main` to break.

▶ Show Sample Solution

CHAPTER 8 OF 12

Topic 8 — Arrays

📚 Topic 8 — Arrays

Bash supports two kinds of arrays: indexed arrays (numbered from zero, like lists) and associative arrays (key-value pairs, like dictionaries). Both let you store multiple values in a single variable and iterate or look them up efficiently. This chapter covers creating, reading, modifying, and deleting array elements; slicing and copying arrays; sorting; splitting strings into arrays; and the important patterns for passing arrays in and out of functions.

1 — Indexed Arrays

An indexed array is a zero-based numbered list of values. Elements can be added, read, updated, or removed individually.

Creating and populating indexed arrays

```
#!/bin/bash

# Method 1: assign all elements at once with ( )
fruits=( apple banana cherry mango )

# Method 2: declare first, then assign individually
declare -a colours
colours[0]="red"
colours[1]="green"
colours[2]="blue"

# Method 3: build from command output
files=( $(ls /etc/*.conf) )    # caution: word-splits on spaces in names
files=()
while IFS= read -r -d '' f; do    # safer: null-delimited via find
    files+=( "$f" )
done < <(find /etc -name "*.conf" -print0)
```

Reading Array Elements

Accessing elements by index

```
fruits=( apple banana cherry mango )

# Single element — MUST use curly braces
echo "${fruits[0]}"            → apple
echo "${fruits[2]}"            → cherry

# Last element
echo "${fruits[-1]}"           → mango    (bash 4.2+)
echo "${fruits[${#fruits[@]}-1]}" → mango (portable)

# All elements as separate words — use in loops and commands
echo "${fruits[@]}"            → apple banana cherry mango

# All elements as a single string (joined by first char of IFS)
echo "${fruits[*]}"            → apple banana cherry mango

# Number of elements
echo "${#fruits[@]}"           → 4

# Length of a specific element
echo "${#fruits[1]}"           → 6  (length of "banana")

# All indices (useful when array may have gaps)
echo "${!fruits[@]}"           → 0 1 2 3
```

Always use "${arr[@]}" (not ${arr[*]}) in loops and command arguments — it keeps elements with spaces intact as separate items.

⚠ **$fruits without brackets is not an array.** Writing `$fruits` (no brackets) is equivalent to `${fruits[0]}` — it gives only the first element and silently discards the rest. Always use `${fruits[@]}` to mean "all elements".

Slicing an Array

Extracting a sub-range of elements

```
letters=( a b c d e f g )

# ${arr[@]:offset:length}
echo "${letters[@]:2:3}"       → c d e   (3 elements from index 2)
echo "${letters[@]:4}"         → e f g   (from index 4 to end)
echo "${letters[@]: -2}"       → f g     (last 2 elements)

# Copy a slice into a new array
middle=( "${letters[@]:2:3}" )
echo "${middle[@]}"            → c d e
```

2 — Modifying Arrays

Append, update, and remove elements

```
arr=( one two three )

# Append one element
arr+=( four )
echo "${arr[@]}"              → one two three four

# Append multiple elements
arr+=( five six )
echo "${arr[@]}"              → one two three four five six

# Update a specific element
arr[1]="TWO"
echo "${arr[@]}"              → one TWO three four five six

# Remove an element by index — leaves a gap (sparse array)
unset 'arr[2]'
echo "${arr[@]}"              → one TWO four five six
echo "${!arr[@]}"             → 0 1 3 4 5  (index 2 is missing!)

# Re-index after deletion to close gaps
arr=( "${arr[@]}" )
echo "${!arr[@]}"             → 0 1 2 3 4  (gap closed)

# Remove the entire array
unset arr
```

Concatenating Arrays

Merging arrays together

```
a=( one two three )
b=( four five six )

# Merge by expanding both into a new array
combined=( "${a[@]}" "${b[@]}" )
echo "${combined[@]}"
one two three four five six

# Prepend elements
a=( zero "${a[@]}" )
echo "${a[@]}"
zero one two three
```

3 — Iterating Over Arrays

Loop patterns for arrays

```
planets=( Mercury Venus Earth Mars Jupiter Saturn )

# Pattern 1: iterate over values (most common)
for planet in "${planets[@]}"; do
    echo "Planet: $planet"
done

# Pattern 2: iterate over indices (needed when index matters)
for i in "${!planets[@]}"; do
    printf "%d: %s\n" "$i" "${planets[$i]}"
done
0: Mercury
1: Venus
...

# Pattern 3: C-style — fine when array has no gaps
for (( i=0; i<${#planets[@]}; i++ )); do
    echo "${planets[$i]}"
done

# Pattern 4: iterate over a sparse array safely (use indices)
sparse=()
sparse[0]="first"
sparse[5]="sixth"
sparse[10]="eleventh"
for i in "${!sparse[@]}"; do
    echo "[$i] = ${sparse[$i]}"
done
[0] = first
[5] = sixth
[10] = eleventh
```

4 — Useful Array Operations

Sorting

Bash has no built-in array sort — use `sort` and capture the output.

Sorting an array

```
names=( Charlie Alice Dave Bob Eve )

# Sort alphabetically
sorted=( $(printf "%s\n" "${names[@]}" | sort) )
echo "${sorted[@]}"
Alice Bob Charlie Dave Eve

# Sort in reverse
rsorted=( $(printf "%s\n" "${names[@]}" | sort -r) )

# Sort numerically
nums=( 20 3 100 15 7 )
nsorted=( $(printf "%s\n" "${nums[@]}" | sort -n) )
echo "${nsorted[@]}"
3 7 15 20 100
```

The `$( printf ... | sort )` approach uses word splitting to rebuild the array, so it breaks on element values containing spaces. For elements with spaces, pipe through `sort` using null delimiters or use a different approach.

Removing Duplicates

Deduplicating an array

```
tags=( bash linux bash python linux shell python )

# Sort and deduplicate with sort -u
unique=( $(printf "%s\n" "${tags[@]}" | sort -u) )
echo "${unique[@]}"
bash linux python shell

# Deduplicate while preserving original order (using associative array as a set)
declare -A _seen
ordered_unique=()
for tag in "${tags[@]}"; do
    if [[ -z "${_seen[$tag]+x}" ]]; then
        _seen["$tag"]=1
        ordered_unique+=( "$tag" )
    fi
done
echo "${ordered_unique[@]}"
bash linux python shell   # first-seen order preserved
```

Searching an Array

Checking if a value exists in an array

```
allowed=( read write execute admin )

# Search function — returns 0 if found, 1 if not
in_array() {
    local needle="$1"; shift
    local item
    for item in "$@"; do
        [[ "$item" == "$needle" ]] && return 0
    done
    return 1
}

if in_array "write" "${allowed[@]}"; then
    echo "'write' is allowed"
fi
'write' is allowed

if ! in_array "delete" "${allowed[@]}"; then
    echo "'delete' is NOT in the list"
fi
```

5 — Associative Arrays

Associative arrays (bash 4.0+) use arbitrary string keys instead of integers. They behave like dictionaries or hash maps in other languages.

Creating and using associative arrays

```
#!/bin/bash

# Must use declare -A — this is not optional
declare -A capitals

# Assign key-value pairs
capitals["France"]="Paris"
capitals["Japan"]="Tokyo"
capitals["Hungary"]="Budapest"
capitals["UK"]="London"

# Or all at once:
declare -A capitals=(
    ["France"]="Paris"
    ["Japan"]="Tokyo"
    ["Hungary"]="Budapest"
    ["UK"]="London"
)

# Read a value by key
echo "Capital of France : ${capitals[France]}"
Capital of France : Paris

# Number of entries
echo "Count: ${#capitals[@]}"
Count: 4

# All values
echo "${capitals[@]}"
Paris Tokyo Budapest London    (order is not guaranteed)

# All keys
echo "${!capitals[@]}"
France Japan Hungary UK
```

Associative arrays do not have a defined order — the iteration order of keys can change between bash versions and runs. If order matters, maintain a separate indexed array of keys.

Iterating Over Associative Arrays

Looping over key-value pairs

```
declare -A scores=( [Alice]=92 [Bob]=78 [Carol]=85 [Dave]=91 )

# Iterate over keys, access values
for name in "${!scores[@]}"; do
    printf "%-10s %d\n" "$name" "${scores[$name]}"
done

# Sorted by key
for name in $(printf "%s\n" "${!scores[@]}" | sort); do
    printf "%-10s %d\n" "$name" "${scores[$name]}"
done
Alice      92
Bob        78
Carol      85
Dave       91
```

Checking Whether a Key Exists

Key existence check

```
declare -A config=( [host]="localhost" [port]="8080" )

# ${var+x} expands to "x" if the key exists, empty if not
if [[ -n "${config[host]+x}" ]]; then
    echo "host is set: ${config[host]}"
fi
host is set: localhost

if [[ -z "${config[timeout]+x}" ]]; then
    echo "timeout key does not exist"
fi
timeout key does not exist

# Delete a key
unset 'config[port]'
echo "${!config[@]}"
host
```

6 — Splitting Strings into Arrays

Two common techniques turn a delimited string into an array — `read -a` with a here-string, and IFS-based word splitting.

String-to-array conversion

```
# read -a with a here-string — split on IFS
csv="apple,banana,cherry"
IFS=',' read -r -a items <<< "$csv"
echo "${items[@]}"                  → apple banana cherry
echo "${items[1]}"                  → banana
echo "${#items[@]}"                 → 3

# Split a colon-delimited string (like PATH)
IFS=':' read -r -a path_dirs <<< "$PATH"
for dir in "${path_dirs[@]}"; do
    echo "$dir"
done

# Split on whitespace using ( $(...) ) — quick but breaks on spaces in values
words=( $(echo "one two three") )
echo "${#words[@]}"                 → 3

# Convert a multi-line string to an array (one line per element)
multiline="first line
second line
third line"
IFS=$'\n' read -r -d '' -a lines <<< "$multiline"
echo "${#lines[@]}"                 → 3
echo "${lines[1]}"                  → second line
```

7 — Arrays and Functions

Bash does not pass arrays to functions directly — when you write `func "${my_array[@]}"`, the function receives a flat list of arguments, not an array object. There are three clean patterns for working around this.

Pattern 1: pass elements as arguments

```
sum_array() {
    local total=0
    for val in "$@"; do
        (( total += val ))
    done
    echo "$total"
}

numbers=( 5 10 15 20 )
total=$(sum_array "${numbers[@]}")
echo "Total: $total"
Total: 50
# Works well when the function only needs to read the values.
# Limitation: you cannot pass two arrays this way without a separator.
```

Pattern 2: pass the array name, use nameref (bash 4.3+)

```
print_array() {
    local -n _arr="$1"       # nameref — _arr IS the caller's array
    local i
    for i in "${!_arr[@]}"; do
        printf "  [%s] %s\n" "$i" "${_arr[$i]}"
    done
}

fruits=( apple banana cherry )
print_array fruits           # pass the NAME, not ${fruits[@]}
  [0] apple
  [1] banana
  [2] cherry

# Works for both indexed and associative arrays.
declare -A config=( [host]="localhost" [port]="8080" )
print_array config
```

Pattern 3: return an array via nameref

```
get_even_numbers() {
    local -n _result="$1"    # output array — write via nameref
    local -i max="$2"
    _result=()               # clear it first
    for (( i=2; i<=max; i+=2 )); do
        _result+=( "$i" )
    done
}

get_even_numbers evens 20
echo "${evens[@]}"
2 4 6 8 10 12 14 16 18 20
```

8 — Practical Examples

Associative array as a config file parser

Read a simple KEY=VALUE config file into an associative array.

```
#!/bin/bash
declare -A cfg

# Read config.ini:  host=localhost  port=8080  debug=true
while IFS='=' read -r key val; do
    [[ "$key" == \#* || -z "$key" ]] && continue   # skip comments/blanks
    cfg["${key// /}"]="${val// /}"                  # trim spaces from key/val
done < config.ini

echo "Host : ${cfg[host]}"
echo "Port : ${cfg[port]}"
```

Stack using an indexed array

Arrays naturally implement push/pop stack behaviour.

```
stack=()

push() { stack+=( "$1" ); }

pop() {
    local -n _out="$1"
    [[ "${#stack[@]}" -eq 0 ]] && { echo "Stack empty" >&2; return 1; }
    _out="${stack[-1]}"
    unset 'stack[-1]'
}

push "first"
push "second"
push "third"

pop item; echo "Popped: $item"
Popped: third
pop item; echo "Popped: $item"
Popped: second
```

9 — Quick Reference

Indexed Arrays

SYNTAX	WHAT IT DOES
`arr=(a b c)`	Create indexed array
`declare -a arr`	Declare (empty) indexed array
`${arr[n]}`	Element at index n
`${arr[-1]}`	Last element (bash 4.2+)
`${arr[@]}`	All elements (each properly quoted)
`${arr[*]}`	All elements joined as one string
`${#arr[@]}`	Number of elements
`${!arr[@]}`	All indices
`${arr[@]:i:n}`	Slice: n elements starting at index i

SYNTAX	WHAT IT DOES
`arr+=(x y)`	Append element(s)
`arr[n]=val`	Set/update element at index n
`unset 'arr[n]'`	Remove element (leaves sparse gap)
`arr=("${arr[@]}")`	Re-index to close sparse gaps
`unset arr`	Delete the entire array

Associative Arrays

SYNTAX	WHAT IT DOES
`declare -A map`	Declare associative array (required)
`map[key]=val`	Set a key-value pair
`${map[key]}`	Read a value by key
`${map[@]}`	All values
`${!map[@]}`	All keys
`${#map[@]}`	Number of entries
`${map[key]+x}`	Non-empty if key exists
`unset 'map[key]'`	Remove a key

🖍️ Exercises

Apply what you have learned in this chapter. Try each exercise yourself before looking at the sample solution.

EXERCISE 1

Write a script called `word_count.sh` that reads a sentence from the user, splits it into an array of words, prints the total word count, lists each word with its index, and then prints the unique words in alphabetical order.

Hint: use `read -r -a words` to split on spaces. Use `${!words[@]}` for indices. Pipe `"${words[@]}"` through `printf "%s\n" | sort -u` for unique sorted words.

▶ Show Sample Solution

EXERCISE 2

Write a script called `phone_book.sh` that uses an associative array to store names and phone numbers. The script should support three operations via command-line arguments: `add NAME NUMBER`, `lookup NAME`, and `list` (prints all entries sorted by name). Store the data in a plain text file (`phonebook.dat`) between runs by writing and reading the array to/from it.

Hint: save to file with `declare -p phonebook > phonebook.dat` and restore with `source phonebook.dat`. Use a `case` statement on `$1` for the three operations. Check if the file exists before sourcing.

▶ Show Sample Solution

EXERCISE 3

Write a script called `stats.sh` that accepts a list of numbers as command-line arguments, stores them in an array, and calculates and prints: the count, the sum, the minimum, the maximum, and the mean (to 2 decimal places). Test with: `./stats.sh 15 3 42 8 27 19 6`

Hint: loop over `"$@"` to build the array and accumulate the sum. Track min and max by comparing each element. Use `bc` for the mean division.

▶ Show Sample Solution

EXERCISE 4

Write a script called `inventory.sh` that uses two parallel associative arrays — one mapping item names to quantities, another mapping item names to unit prices — to manage a simple inventory. Implement `add`, `sell` (reduce quantity), and `report` commands. The report should print a formatted table showing each item, its quantity, unit price, and total value, plus a grand total at the bottom.

Hint: use `declare -A qty` and `declare -A price`. Persist both with `declare -p`. For the report, iterate over sorted keys of `qty` and multiply `${qty[$item]}` by `${price[$item]}` using `bc`.

▶ Show Sample Solution

CHAPTER 9 OF 12

Topic 9 — Working with Files and Text

📁 Topic 9 — Working with Files and Text

The shell's real power comes from combining simple text-processing tools into pipelines that transform data. This chapter covers reading and writing files safely, navigating the filesystem with `find`, and the essential Unix text tools — `grep`, `sed`, `awk`, `cut`, `sort`, `uniq`, `wc`, and `tr` — with an emphasis on the patterns you'll actually use in scripts every day.

1 — Reading Files

The canonical, safe way to read a file line by line is a `while IFS= read -r` loop. It handles blank lines, lines without a trailing newline, and filenames or values that contain spaces.

Reading a file line by line

```
#!/bin/bash

# Canonical pattern — handles all edge cases
while IFS= read -r line; do
    echo "Line: $line"
done < "/path/to/file.txt"

# With line numbers
lineno=0
while IFS= read -r line; do
    (( lineno++ ))
    printf "%4d  %s\n" "$lineno" "$line"
done < file.txt

# Skip blank lines and comments (lines starting with #)
while IFS= read -r line; do
    [[ -z "$line" || "$line" == \#* ]] && continue
    echo "$line"
done < config.txt

# Read two fields per line (e.g. "name score" format)
while read -r name score; do
    printf "%-15s %d\n" "$name" "$score"
done < scores.txt
```

IFS= prevents leading/trailing whitespace being stripped from each line. -r prevents backslash escape sequences being interpreted. Both are almost always what you want.

⚠ **Don't do: `for line in $(cat file)`**

This splits on every whitespace character (not just newlines), breaks on filenames with spaces, and is slower than a `while read` loop. Always use the `while IFS= read -r` pattern for processing files line by line.

Reading a File into a Variable or Array

Slurp an entire file or split into an array

```
# Read entire file into a single variable
content=$(<file.txt)          # faster than $(cat file.txt)

# Read all lines into an array (one element per line)
lines=()
while IFS= read -r line; do
    lines+=( "$line" )
done < file.txt

# Or with mapfile / readarray (bash 4+, most concise)
mapfile -t lines < file.txt
# -t strips the trailing newline from each element

echo "Total lines: ${#lines[@]}"
echo "First line : ${lines[0]}"
echo "Last line  : ${lines[-1]}"
```

2 — Writing Files

Output redirection patterns

```
# Overwrite (create or truncate)
echo "Hello" > output.txt

# Append
echo "World" >> output.txt

# Write multiple lines with a here-document
cat > config.ini <<'EOF'
host=localhost
port=8080
debug=false
EOF

# Write with variable expansion in here-doc (no quotes on delimiter)
app_name="myapp"
version="1.0"
cat > version.txt <<EOF
Application: $app_name
Version    : $version
Built      : $(date '+%Y-%m-%d')
EOF

# Write stdout AND stderr to the same log file
logfile="app.log"
{
    echo "Starting process..."
    some_command
    echo "Done."
} &> "$logfile"

# Atomic write — write to temp file first, then rename
# (prevents partial reads if another process opens the file mid-write)
tmpfile=$(mktemp)
generate_data > "$tmpfile"
mv "$tmpfile" "final_output.txt"
```

Tip — atomic writes with mktemp: *When a script updates a file that another process might be reading (like a status file or config), always write to a temporary file first and then `mv` it into place. A rename on the same filesystem is atomic; a plain `>` redirect is not — a reader could see a half-written file.*

3 — Finding Files with `find`

`find` is the standard tool for locating files by name, type, size, age, permissions, or any combination. It recursively traverses the directory tree and can execute actions on matching files.

find — common patterns

```
# Find by name (case-sensitive)
find /var/log -name "*.log"

# Find by name (case-insensitive)
find . -iname "*.jpg"

# Find only files (not directories)
find . -type f -name "*.sh"

# Find only directories
find . -type d -name "config"

# Find files modified in the last 7 days
find . -type f -mtime -7

# Find files larger than 100 MB
find / -type f -size +100M

# Limit search depth (don't recurse deeper than 2 levels)
find . -maxdepth 2 -name "*.conf"

# Run a command on each found file (-exec ... {} \;)
find . -name "*.sh" -exec chmod +x {} \;

# Safer: use -print0 | xargs -0 to handle spaces in names
find . -name "*.log" -print0 | xargs -0 rm -f

# Delete empty directories
find . -type d -empty -delete

# Find and loop in bash (safest — handles all filenames)
while IFS= read -r -d '' file; do
    echo "Processing: $file"
done <(find . -name "*.txt" -print0)
```

The `-print0` + `read -d ''` combination uses null bytes as the record separator, making it safe for filenames that contain spaces, newlines, or other special characters.

4 — Searching Text with grep

`grep` searches for lines matching a pattern. In scripts you use it both to filter output in pipelines and to test whether a match exists at all (via its exit code).

grep patterns

```
# Basic search — print matching lines
grep "error" app.log

# Case-insensitive
grep -i "error" app.log

# Show line numbers
grep -n "TODO" *.py

# Invert match — lines that do NOT match
grep -v "^#" config.txt          # strip comment lines

# Count matching lines
grep -c "FAIL" results.txt

# Show only the matched part, not the whole line
grep -o "[0-9]\+\.[0-9]\+\.[0-9]\+\.[0-9]\+" access.log  # extract IPs

# Extended regex (no need to escape + ? | ( ) )
grep -E "^(ERROR|WARN)" app.log

# Recursive search in a directory tree
grep -r "password" /etc/

# Show N lines of context before/after the match
grep -A 3 "Exception" app.log    # 3 lines after
grep -B 2 "Exception" app.log    # 2 lines before
grep -C 2 "Exception" app.log    # 2 lines either side

# Use exit code in a script (0 = found, 1 = not found)
if grep -q "CRITICAL" app.log; then   # -q = quiet, no output
    echo "Critical errors found!" >&2
    exit 1
fi
```

5 — Stream Editing with sed

`sed` (stream editor) processes text line by line, making substitutions, deletions, and other edits. The substitution command `s/pattern/replacement/` is by far the most used.

sed — substitution and deletion

```
# Replace first occurrence on each line
sed 's/foo/bar/' input.txt

# Replace ALL occurrences on each line (g = global)
sed 's/foo/bar/g' input.txt

# Case-insensitive replacement
sed 's/error/ERROR/gI' app.log

# Edit in place (modify the file directly)
sed -i 's/localhost/192.168.1.1/g' config.ini
# -i.bak makes a backup: config.ini.bak
sed -i.bak 's/localhost/192.168.1.1/g' config.ini

# Delete lines matching a pattern
sed '/^#/d' config.txt             # delete comment lines
sed '/^[[:space:]]*$/d' file.txt   # delete blank lines

# Print only specific lines (suppress default output with -n)
sed -n '5p' file.txt               # print line 5
sed -n '5,10p' file.txt            # print lines 5-10
sed -n '/START/,/END/p' file.txt   # print between markers

# Multiple expressions with -e
sed -e 's/foo/bar/g' -e 's/baz/qux/g' file.txt

# Use & to refer to the whole matched text
sed 's/[0-9]\+/[&]/g' file.txt    # wrap every number in brackets
Price [42] for [5] items

# Strip leading and trailing whitespace
sed 's/^[[:space:]]*//; s/[[:space:]]*$//' file.txt
```

⚠ **macOS sed vs GNU sed:** On macOS, `-i` requires an explicit backup suffix — `sed -i '' 's/a/b/' file` (empty string). On Linux, `sed -i 's/a/b/' file` works without a suffix. For portability in scripts, use `-i.bak` (creates a backup that you can then delete).

6 — Field Processing with awk

awk splits each input line into fields and lets you apply rules to each line. It's ideal for columnar data: log files, CSV, /etc/passwd, command output.

awk — essential patterns

```
# Built-in variables:
#   $0  — entire line        $1 $2 ... — individual fields
#   NR  — current line number NF — number of fields on this line
#   FS  — field separator    OFS — output field separator

# Print specific fields (default delimiter: any whitespace)
awk '{print $1, $3}' data.txt

# Print last field
awk '{print $NF}' data.txt

# Use a custom field separator
awk -F: '{print $1, $3}' /etc/passwd     # username and UID
awk -F, '{print $2}' data.csv

# Print lines where a field matches a pattern
awk '/ERROR/ {print NR, $0}' app.log
awk '$3 > 100 {print $1, $3}' scores.txt   # numeric comparison

# Sum a column
awk '{sum += $2} END {print "Total:", sum}' sales.txt

# Count lines matching a pattern
awk '/FAIL/ {count++} END {print count " failures"}' results.txt

# BEGIN and END blocks run before/after all input
awk 'BEGIN {print "Name", "Score"} {print $1, $2} END {print "Done"}' scores.txt

# Reformatting: change delimiter in output
awk -F, 'BEGIN {OFS="|"} {print $1, $2, $3}' data.csv

# Capture awk output in a variable
total=$(awk '{sum += $1} END {print sum}' numbers.txt)
echo "Total: $total"
```

7 — The Supporting Cast: cut, sort, uniq, wc, tr

cut — extract columns

cut

```
# Cut by delimiter and field number
cut -d: -f1 /etc/passwd             # usernames
cut -d, -f2,4 data.csv              # columns 2 and 4
cut -d, -f2- data.csv               # columns 2 to end

# Cut by character position
cut -c1-8 timestamps.txt            # first 8 characters
cut -c9- timestamps.txt             # from character 9 to end
```

sort — order lines

sort

```
sort names.txt                      # alphabetical
sort -r names.txt                   # reverse alphabetical
sort -n numbers.txt                 # numeric
sort -nr numbers.txt                # numeric descending (largest first)
sort -u names.txt                   # sort and remove duplicates
sort -t, -k2,2n data.csv            # sort CSV by 2nd column numerically
sort -k1,1 -k2,2n data.txt          # primary sort col 1, secondary col 2
sort -h sizes.txt                   # human-numeric (10K before 2M)
```

uniq — remove adjacent duplicate lines

uniq (input must be sorted first)

```
sort items.txt | uniq               # remove duplicates
sort items.txt | uniq -c            # prefix each line with its count
sort items.txt | uniq -d            # print only lines that appeared more than once
sort items.txt | uniq -u            # print only lines that appeared exactly once

# Top 10 most frequent lines
sort access.log | uniq -c | sort -rn | head -10
```

wc — count lines, words, characters

wc

```
wc -l file.txt                    # number of lines
wc -w file.txt                    # number of words
wc -c file.txt                    # number of bytes
wc -m file.txt                    # number of characters (multi-byte aware)

# Capture line count cleanly in a variable
count=$(wc -l < file.txt)           # redirect avoids filename in output
echo "Lines: $count"
```

tr — translate or delete characters

tr (reads from stdin only)

```
# Convert to uppercase / lowercase
echo "hello world" | tr '[:lower:]' '[:upper:]'
HELLO WORLD
echo "HELLO" | tr 'A-Z' 'a-z'
hello

# Delete specific characters
echo "h3ll0 w0rld" | tr -d '0-9'
hll wrld

# Squeeze repeated characters (e.g. collapse multiple spaces)
echo "too   many   spaces" | tr -s ' '
too many spaces

# Replace colons with newlines (e.g. expand PATH for readability)
echo "$PATH" | tr ':' '\n'

# Remove Windows carriage returns from a file
tr -d '\r' < windows.txt > unix.txt
```

8 — Building Pipelines

The real power is combining these tools. Each tool does one job well; the pipe | connects them into a transformation chain.

Pipeline: top 5 IP addresses in an Apache log

```
awk '{print $1}' access.log \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -5
    523 192.168.1.105
    311 10.0.0.22
    198 172.16.0.4
    145 192.168.1.200
     89 10.0.0.1
```

Pipeline: extract failed logins from auth.log

```
grep "Failed password" /var/log/auth.log \
  | awk '{print $(NF-3)}' \
  | sort | uniq -c | sort -rn \
  | head -10
```

Pipeline: CSV summary — total sales per region

```
# Input: date,region,amount  e.g.  2026-01-05,North,1500
tail -n +2 sales.csv \          # skip header
  | awk -F, '{region[$2] += $3}
        END {for (r in region) printf "%-10s £%d\n", r, region[r]}' \
  | sort -k2,2rn

South      £48200
North      £35700
East       £29100
```

tee — split a pipeline to a file and stdout

tee

```
# Log pipeline output to a file while still printing to screen
some_command | tee output.log | grep "ERROR"

# Append with -a
some_command | tee -a logfile.log >/dev/null  # log only, suppress screen
```

9 — Process Substitution

Process substitution — `<(command)` — lets you feed the output of a command to another command that expects a filename. It's the clean way to use `diff`, `while read`, and other tools with live command output.

Process substitution patterns

```
# diff two commands' output without temp files
diff <(sort file1.txt) <(sort file2.txt)

# Read the output of a command safely in a while loop
# (a plain pipe would run the loop body in a subshell)
while IFS= read -r line; do
    echo "$line"
done <(grep "ERROR" app.log)

# Compare sorted lists from two directories
diff <(ls dir1/) <(ls dir2/)

# Write to a process (less common)
tee >(gzip > backup.gz) > plain_copy.txt < source.txt
```

The key advantage over a pipe in a `while` loop: variables set inside the loop body remain visible after the loop ends, because `<()` runs the command in a separate process but keeps the `while` loop in the current shell.

10 — Quick Reference

TOOL / PATTERN	WHAT IT DOES	KEY FLAGS
`while IFS= read -r line; do ... done < file`	Safe line-by-line file reading	`-r` no backslash processing

TOOL / PATTERN	WHAT IT DOES	KEY FLAGS
`mapfile -t arr < file`	Read all lines into an array	`-t` strips trailing newline
`content=$(<file)`	Slurp whole file into variable	—
`find dir -name "*.ext" -type f`	Locate files recursively	`-mtime -7`, `-size +100M`, `-exec`, `-print0`
`grep -E "pattern" file`	Print matching lines	`-i` case-insensitive, `-v` invert, `-q` silent, `-c` count, `-n` line numbers
`sed 's/old/new/g' file`	Stream substitution	`-i` in-place, `-n` suppress output, `/d` delete lines
`awk -F: '{print $1}' file`	Field extraction / processing	`NR` line no., `NF` field count, `BEGIN/END`
`cut -d, -f2 file`	Extract columns by delimiter	`-c` character positions
`sort -n -k2 file`	Sort lines	`-r` reverse, `-u` unique, `-h` human sizes
`uniq -c`	Remove adjacent duplicates / count	`-d` duplicates only, `-u` unique only
`wc -l < file`	Count lines (words, bytes)	`-w` words, `-c` bytes, `-m` chars
`tr 'a-z' 'A-Z'`	Translate characters	`-d` delete, `-s` squeeze repeats
`tee file`	Copy stdin to file and stdout	`-a` append
`diff <(cmd1) <(cmd2)`	Compare command output with process substitution	—

🖍️ Exercises

Apply what you have learned. Try writing the script yourself before looking at the sample solution.

EXERCISE 1

Write a script called `log_report.sh` that accepts a log file as its first argument and prints: (a) total number of lines, (b) number of lines containing `ERROR`, (c) number of lines containing `WARN`, and (d) the 5 most frequent words in ERROR lines, with their counts.

Hint: use `wc -l < file` for counts, `grep -c` for pattern counts, and `grep "ERROR" | tr -s ' ' '\n' | sort | uniq -c | sort -rn | head -5` for frequent words.

▶ Show Sample Solution

EXERCISE 2

Write a script called `csv_filter.sh` that reads a CSV file (with a header row), takes a column number and a search term as arguments, and prints all rows where that column matches the term. Also print the header. Example: `./csv_filter.sh sales.csv 2 North` prints all rows where column 2 is "North".

Hint: use `head -1` to print the header, then `tail -n +2` to skip it and pipe to `awk -F,` with a condition on `$colnum`.

▶ Show Sample Solution

EXERCISE 3

Write a script called `find_large.sh` that accepts a directory and a size threshold (in MB) as arguments, finds all files larger than that threshold, and outputs a formatted table showing the filename (base name only) and size in MB, sorted largest first. At the end, print the total size of all matched files.

Hint: use `find dir -type f -size +NNMb` (or use `+NNM`). Pipe to `du -m` or use `stat` to get sizes. Collect results into an array, sort with `sort -rn`, and sum with `awk`.

▶ Show Sample Solution

EXERCISE 4

Write a script called `replace_in_files.sh` that accepts three arguments: a directory, a search string, and a replacement string. It should find all `.txt` files in that directory tree containing the search string, report how many files were found, show a preview of the first match in each file, and then (after confirmation) perform the replacement in all files using `sed -i`. Make a `.bak` backup of each file before modifying it.

Hint: use `grep -rl` to find files containing the pattern. Use `grep -m1` for a single-line preview. Use `read -r -p "Proceed? [y/N]"` for confirmation. Use `sed -i.bak` for atomic in-place replacement with backup.

▶ Show Sample Solution

CHAPTER 10 OF 12

Topic 10 — Pattern Matching and Regular Expressions

🔍 Topic 10 — Pattern Matching and Regular Expressions

Bash uses two related but distinct pattern systems. Glob patterns (also called shell patterns) are used for filename matching and the `case` statement — they use `*`, `?`, and `[...]`. Regular expressions (regex) are used inside `[[ =~ ]]` and tools like `grep`, `sed`, and `awk` — they are far more expressive. This chapter covers both systems thoroughly, explains where each one applies, and shows the key patterns you'll reach for constantly in real scripts.

1 — Glob Patterns (Shell Wildcards)

Globs are expanded by the shell itself before any command sees them. They match filenames in the filesystem — or strings in `[[ == ]]` and `case`.

PATTERN	MATCHES	EXAMPLE
`*`	Any string of zero or more characters (not including `/`)	`*.log` → all `.log` files
`?`	Exactly one character	`file?.txt` → `file1.txt`, `fileA.txt`
`[abc]`	One character from the set	`[abc].sh` → `a.sh`, `b.sh`, `c.sh`
`[a-z]`	One character in the range	`[0-9].txt` → single-digit filenames
`[^abc]`	One character NOT in the set	`[^0-9]*` → files not starting with a digit
`**`	Any path including `/` (requires `globstar` option)	`**/*.py` → all `.py` files recursively

Glob patterns in action

```
# Filename expansion (pathname expansion)
ls *.sh                         # all shell scripts
ls report_2026-??.csv            # report_2026-01.csv through 09.csv etc.
ls [A-Z]*.txt                   # text files starting with a capital letter

# Recursive glob — must enable globstar first
shopt -s globstar
for f in **/*.py; do
    echo "$f"
done

# Include dotfiles (hidden files) in globs
shopt -s dotglob
ls *                            # now includes .hidden files

# Return glob unexpanded if no match (instead of passing literal string)
# nullglob: expands to nothing if no match  failglob: throws error
shopt -s nullglob
files=( *.csv )
[[ "${#files[@]}" -eq 0 ]] && echo "No CSV files found"

# Globs in case — match strings, not filenames
filename="archive.tar.gz"
case "$filename" in
    *.tar.gz)  echo "gzipped tarball" ;;
    *.zip)     echo "zip archive" ;;
    *.sh)      echo "shell script" ;;
    *)         echo "unknown type" ;;
esac
gzipped tarball
```

⚠ **Always quote glob results when assigning to variables.** `file=*.txt` stores the literal string `*.txt`. `files=(*.txt)` correctly expands into an array. When looping: `for f in *.txt` is fine unquoted, but once the filename is in a variable, use `"$f"` everywhere.

2 — Extended Globs

Extended globs add five powerful pattern operators. Enable them with `shopt -s extglob`. They work in filename expansion, `case`, and `[[ == ]]`.

PATTERN	MEANING	EXAMPLE
`?(pat)`	Zero or one occurrence of *pat*	`file?(s).txt` → `file.txt` or `files.txt`
`*(pat)`	Zero or more occurrences of *pat*	`*(0)1` → `1`, `01`, `001`
`+(pat)`	One or more occurrences of *pat*	`+([0-9])` → one or more digits
`@(pat)`	Exactly one occurrence of *pat* (alternation)	`@(jpg|png|gif)` → exactly one of those
`!(pat)`	Anything except *pat*	`!(*.log)` → all files except `.log`

Extended glob examples

```
shopt -s extglob

# Match image files with one extension from a list
for img in *.@(jpg|jpeg|png|gif|webp); do
    echo "Image: $img"
done

# List everything EXCEPT backup files
ls !(*.bak|*.tmp)

# Match version strings: v1, v12, v123 (but not v)
ver="v42"
[[ "$ver" == v+([0-9]) ]] && echo "valid version"
valid version

# Strip extension using extended glob in parameter expansion
file="photo.backup.tar.gz"
echo "${file%%+(.+([a-z]))}"      # strip all dot-extensions
photo

# case with extended globs
input="yes"
case "$input" in
    @(y|yes|Y|YES))  echo "Confirmed" ;;
    @(n|no|N|NO))    echo "Declined" ;;
    *)               echo "Unknown response" ;;
esac
```

3 — Pattern Matching Inside [[ ]]

The double-bracket `[[ ]]` construct supports two kinds of matching: glob patterns with `==`, and regular expressions with `=~`.

[[==]] glob matching vs [[=~]] regex matching

```
filename="report_2026-06.csv"

# == uses a GLOB pattern (not regex) — right side is unquoted
[[ "$filename" == report_*.csv ]]  && echo "glob match"
glob match

# If you QUOTE the pattern it becomes a literal string comparison
[[ "$filename" == "report_*.csv" ]] && echo "this won't print — literal * char"

# =~ uses EXTENDED REGEX — right side is also unquoted
[[ "$filename" =~ report_[0-9]{4}-[0-9]{2}\.csv ]] && echo "regex match"
regex match

# Practical: validate an IPv4 address
ip="192.168.1.105"
octet='([0-9]{1,3})'
if [[ "$ip" =~ ^${octet}\.${octet}\.${octet}\.${octet}$ ]]; then
    echo "Looks like an IP address"
fi

# Store the regex in a variable for readability (do NOT quote it)
email_re='^[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,}$'
[[ "test@example.com" =~ $email_re ]] && echo "valid email format"
valid email format
```

Store complex regex patterns in a variable and use `$var` unquoted on the right side of `=~`. Quoting the right side of either `==` or `=~` makes it a literal string comparison — the pattern metacharacters are ignored.

4 — Capturing Groups with `BASH_REMATCH`

After a successful `=~` match, bash populates the read-only array `BASH_REMATCH`: index `[0]` is the whole match, and indices `[1]`, `[2]`… are the captured groups (parenthesised parts of the pattern).

Using BASH_REMATCH to extract parts of a string

```bash
# Extract year, month, day from a date string
date_str="Today is 2026-06-09 and it's Tuesday."
date_re='([0-9]{4})-([0-9]{2})-([0-9]{2})'

if [[ "$date_str" =~ $date_re ]]; then
    echo "Full match : ${BASH_REMATCH[0]}"   → 2026-06-09
    echo "Year       : ${BASH_REMATCH[1]}"   → 2026
    echo "Month      : ${BASH_REMATCH[2]}"   → 06
    echo "Day        : ${BASH_REMATCH[3]}"   → 09
fi

# Parse a URL into components
url="https://api.example.com:8443/v2/users?page=2"
url_re='^(https?)://([^:/]+)(:([0-9]+))?(/[^?]*)(\?.*)?$'

[[ "$url" =~ $url_re ]] && {
    echo "Scheme : ${BASH_REMATCH[1]}"    → https
    echo "Host   : ${BASH_REMATCH[2]}"    → api.example.com
    echo "Port   : ${BASH_REMATCH[4]}"    → 8443
    echo "Path   : ${BASH_REMATCH[5]}"    → /v2/users
    echo "Query  : ${BASH_REMATCH[6]}"    → ?page=2
}
```

5 — Regular Expression Fundamentals

Regex is its own mini-language. Here are the building blocks you need to know. Bash's `=~` uses **Extended Regular Expressions (ERE)** — the same dialect as `grep -E` and `awk`.

ANCHORS	CHARACTER CLASSES	QUANTIFIERS

```
^    start of string
$    end of string
\b   word boundary
     (grep/sed only)

# Match whole string
^hello$   → only "hello"

# Start only
^error    → "error" at star
```

```
[abc]    one of a, b, c
[^abc]   not a, b, or c
[a-z]    lowercase letter
[0-9]    digit
.        any char (not \n)
\d       digit (some tools)

# POSIX classes (portable)
[:alpha:]   letters
[:digit:]   digits
[:space:]   whitespace
[:alnum:]   letters+digits
```

```
?       0 or 1
*       0 or more
+       1 or more
{n}     exactly n
{n,}    n or more
{n,m}   between n and m

# Greedy vs lazy
# Default is greedy (match
.*    greedy
# Lazy not in basic ERE/BR
# use Perl-style grep -P i
```

GROUPS & ALTERNATION

```
(abc)      group / capture
(?:abc)    non-capture group
               (not in BRE)
a|b        a OR b
(cat|dog)  cat OR dog

# Alternation examples
^(ERROR|WARN|INFO)
# matches log level prefix
```

ESCAPING SPECIAL CHARS

```
# In ERE these need escapi
# to be treated as literal
\.  \*  \+  \?  \(  \)
\[  \]  \{  \}  \^  \$  \|

# Match a literal dot
192\.168\.1\.[0-9]+

# Match a literal parenthe
\(deprecated\)
```

USEFUL SHORTCUTS

```
# Integer: one or more dig
[0-9]+    or    [[:digit:]

# Word: letters/digits/und
[a-zA-Z0-9_]+

# Optional sign + integer
-?[0-9]+

# Whitespace (one or more)
[[:space:]]+

# Blank line
^[[:space:]]*$
```

BRE vs ERE — What Changes?

Two regex dialects in Linux:

BRE (Basic Regular Expressions) — used by default `grep` and `sed`. The metacharacters `+ ? | ( ) { }` are *literal* unless you escape them with `\`. So `+` is a literal plus; `\+` means "one or more".

ERE (Extended Regular Expressions) — used by `grep -E`, `awk`, and bash's `=~`. The metacharacters `+ ? | ( ) { }` are *special* by default; you escape with `\` to match them literally.

Rule of thumb: always use ERE. Pass `-E` to `grep` and `-E` to `sed`. Less escaping, more readable.

6 — Regex in grep

grep -E (Extended Regex) patterns

```
# Match lines starting with a log level
grep -E "^(ERROR|WARN|CRITICAL)" app.log

# Extract email addresses from a file
grep -Eo "[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,}" file.txt

# Find lines with a 4-digit year between 1900 and 2099
grep -E "(19|20)[0-9]{2}" dates.txt

# Match lines containing BOTH "error" AND "disk" (chained grep)
grep -i "error" syslog | grep -i "disk"

# Match lines of 10 or more characters
grep -E ".{10,}" file.txt

# Find lines with repeated words ("the the", "is is", etc.)
grep -E "\b([a-z]+) \1\b" essay.txt

# Extract IPv4 addresses
grep -Eo "([0-9]{1,3}\.){3}[0-9]{1,3}" access.log | sort -u

# Lines that do NOT match a pattern
grep -Ev "^(#|$)" config.txt      # exclude comments and blank lines
```

7 — Regex in sed

The `s/pattern/replacement/` command in sed uses BRE by default. Use `sed -E` to switch to ERE (avoiding the need to escape `+`, `()`, etc.).

sed -E (Extended Regex) substitutions

```
# Normalise whitespace (collapse runs of spaces to one)
sed -E 's/[[:space:]]+/ /g' file.txt

# Remove HTML tags
sed -E 's/<[^>]+>//g' page.html

# Capture groups with \1 \2 back-references
# Reformat dates from DD/MM/YYYY to YYYY-MM-DD
echo "09/06/2026" | sed -E 's|([0-9]{2})/([0-9]{2})/([0-9]{4})|\3-\2-\1|'
2026-06-09

# Wrap numbers in brackets using & (whole match)
echo "Item costs 42 pounds" | sed -E 's/[0-9]+/[&]/g'
Item costs [42] pounds

# Swap first and second fields on a colon-delimited line
echo "alice:admin" | sed -E 's/^([^:]+):([^:]+)/\2:\1/'
admin:alice

# Delete lines matching a pattern
sed -E '/^[[:space:]]*(#|$)/d' config.txt   # strip blanks and comments

# Print only lines between two markers
sed -n '/^START/,/^END/p' file.txt
```

8 — Regex in awk

awk natively uses ERE. Patterns can appear as standalone conditions, inside `if`, or with the ~ (match) and !~ (no match) operators on specific fields.

awk regex patterns and operators

```
# Standalone regex — filter lines matching the pattern
awk '/^ERROR/' app.log

# Negate — lines NOT matching
awk '!/^#/' config.txt

# ~ operator: match a specific field
awk -F: '$1 ~ /^[a-z]/ {print $1}' /etc/passwd   # usernames starting lowercase
awk -F, '$3 !~ /[0-9]/ {print $0}' data.csv     # rows where col 3 has no digit

# Extract + reformat using match() and capture
# gawk (GNU awk) supports capture groups in match()
echo "2026-06-09" | gawk 'match($0, /([0-9]{4})-([0-9]{2})-([0-9]{2})/, a) {
    printf "Day: %s, Month: %s, Year: %s\n", a[3], a[2], a[1]
}'
Day: 09, Month: 06, Year: 2026

# Process a range of lines between two patterns
awk '/BEGIN_SECTION/,/END_SECTION/' file.txt

# Conditional with sub() / gsub() for regex replacement
awk '{gsub(/[[:space:]]+/, "_"); print}' file.txt   # spaces → underscores
```

9 — Practical Validation Patterns

These are production-ready regex fragments for common validation tasks in Bash scripts.

Input validation with [[=~]]

```
#!/bin/bash

# ── Integers ──────────────────────────────────────
is_integer() { [[ "$1" =~ ^-?[0-9]+$ ]]; }

# ── Positive integer (no sign) ────────────────────
is_positive_int() { [[ "$1" =~ ^[0-9]+$ ]]; }

# ── Decimal number ────────────────────────────────
is_number() { [[ "$1" =~ ^-?[0-9]+(\.[0-9]+)?$ ]]; }

# ── Email (basic) ─────────────────────────────────
email_re='^[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,}$'
is_email() { [[ "$1" =~ $email_re ]]; }

# ── IPv4 address ──────────────────────────────────
ipv4_re='^([0-9]{1,3}\.){3}[0-9]{1,3}$'
is_ipv4() { [[ "$1" =~ $ipv4_re ]]; }

# ── ISO date YYYY-MM-DD ───────────────────────────
date_re='^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$'
is_iso_date() { [[ "$1" =~ $date_re ]]; }

# ── Hostname ──────────────────────────────────────
host_re='^[a-zA-Z0-9]([a-zA-Z0-9\-]{0,61}[a-zA-Z0-9])?(\.[a-zA-Z]{2,})+$'
is_hostname() { [[ "$1" =~ $host_re ]]; }

# ── Usage ─────────────────────────────────────────
is_integer  "-42"            && echo "integer ✓"
is_email    "me@example.com" && echo "email ✓"
is_iso_date "2026-06-09"     && echo "date ✓"
! is_ipv4   "999.0.0.1"      && echo "bad IP ✓"
```

10 — Quick Reference

Glob vs Regex — side by side

GOAL	GLOB (SHELL, CASE, ==)	REGEX (=~, GREP -E, AWK)
Any string	`*`	`.*`

GOAL	GLOB (SHELL, CASE, ==)	REGEX (=~, GREP -E, AWK)
Any single character	`?`	`.`
One of these characters	`[abc]`	`[abc]`
Start of string	N/A (matches whole string)	`^`
End of string	N/A	`$`
One or more	`+(pat)` extglob	`+`
Zero or one	`?(pat)` extglob	`?`
Alternation	`@(a|b)` extglob	`(a|b)`
Negate	`!(pat)` extglob	`[^...]` or grep `-v`

Regex metacharacter summary (ERE)

SYMBOL	MEANING
`^`	Start of string / line
`$`	End of string / line
`.`	Any single character (except newline)
`*`	Zero or more of preceding
`+`	One or more of preceding
`?`	Zero or one of preceding
`{n,m}`	Between n and m occurrences
`[abc]`	Character class
`[^abc]`	Negated character class
`(abc)`	Capturing group
`a|b`	Alternation — a or b
`\`	Escape next metacharacter
`[:alpha:]`	POSIX letter class (inside `[...]`)
`[:digit:]`	POSIX digit class (inside `[...]`)

SYMBOL	MEANING
`[:space:]`	POSIX whitespace class (inside `[...]`)

Where each pattern system is used

CONTEXT	SYSTEM	NOTES
Filename expansion	Glob	Expanded by shell before command runs
`case` patterns	Glob (+ extglob)	Matches whole string, not substring
`[[ str == pat ]]`	Glob (+ extglob)	Right side must be **unquoted**
`[[ str =~ re ]]`	ERE	Right side unquoted; sets `BASH_REMATCH`
`grep` (default)	BRE	Use `-E` for ERE
`grep -E`	ERE	Recommended for readability
`sed` (default)	BRE	Use `-E` for ERE
`awk`	ERE	Always ERE, no flag needed

🖍️ Exercises

Apply what you have learned. Write each script yourself before looking at the sample solution.

EXERCISE 1

Write a script called `validate_input.sh` that prompts the user for five pieces of information one at a time — name, age, email address, an IPv4 address, and a date in YYYY-MM-DD format — and validates each with a regex using `[[ =~ ]]`. Keep re-prompting for each field until valid input is entered. Print a summary once all five fields are collected.

Hint: write one `while true` loop per field. Use the validation functions from section 9 (is_integer, is_email, is_ipv4, is_iso_date). For the name, require at least 2 alphabetic characters.

▶ Show Sample Solution

EXERCISE 2

Write a script called `parse_log.sh` that reads an Apache-style access log file (path as argument) and uses `BASH_REMATCH` to parse each line. Extract the IP address, HTTP method, URL path, and response code. Print a summary showing: total requests, unique IPs, count of each HTTP method, and count of each response code, sorted numerically.

Hint: Apache combined log format is: `IP - - [date] "METHOD /path HTTP/1.1" CODE size ...`. Build a regex with capture groups for IP (`group 1`), method (`group 2`), path (`group 3`), code (`group 4`). Use associative arrays to accumulate counts.

▶ Show Sample Solution

EXERCISE 3

Write a script called `rename_dated.sh` that renames files in the current directory whose names contain a date in DD-MM-YYYY format to use ISO format (YYYY-MM-DD) instead. For example, `report_09-06-2026.csv` becomes `report_2026-06-09.csv`. Dry-run mode (when called with `--dry-run`) should print what would be renamed without actually doing it.

Hint: use `for f in *` with `[[ "$f" =~ ([0-9]{2})-([0-9]{2})-([0-9]{4}) ]]`. Rebuild the new filename using `BASH_REMATCH` and `sed` or parameter expansion to swap the date portion. Check for a `--dry-run` argument with `$1`.

▶ Show Sample Solution

EXERCISE 4

Write a script called `extract_urls.sh` that accepts a file (HTML or text) and extracts all unique URLs from it using `grep -Eo`. Print them one per line, sorted, with duplicates removed. As a bonus, categorise them: print http/https URLs first, then mailto: links, then any other protocol.

Hint: use `grep -Eo 'https?://[^"<> ]+|mailto:[^"<> ]+'` to extract URLs. Pipe through `sort -u` to deduplicate. Use `grep` to split into categories, or use a loop with `[[ =~ ]]` to classify each URL.

▶ Show Sample Solution

CHAPTER 11 OF 12

Topic 11 — Error Handling and Debugging

🛡️ Topic 11 — Error Handling and Debugging

A script that silently continues past a failed command and corrupts data is far more dangerous than one that crashes loudly. Professional Bash scripts are defensive by design — they set strict execution flags, trap unexpected exits, log what they do, and clean up after themselves no matter how they end. This chapter covers every layer of that defence, plus the full toolkit for finding and fixing bugs when things go wrong.

1 — Exit Codes

Every command in Bash exits with a numeric status code — **0 means success**, any non-zero value means failure. This is the foundation of all error handling.

Reading and using exit codes

```
# $? holds the exit code of the most recent command
ls /tmp
echo "Exit code: $?"          → 0  (success)

ls /nonexistent 2>/dev/null
echo "Exit code: $?"          → 2  (no such file)

# Check exit code immediately after a command
cp source.txt dest.txt
if [[ $? -ne 0 ]]; then
    echo "Copy failed" >&2
    exit 1
fi

# More concise idiom: use the command directly in if
if ! cp source.txt dest.txt; then
    echo "Copy failed" >&2
    exit 1
fi

# Common exit code conventions
exit 0    # success
exit 1    # general error
exit 2    # misuse of shell built-in / bad argument
exit 126  # command found but not executable
exit 127  # command not found
exit 130  # terminated by Ctrl+C (128 + signal 2)
```

⚠ **$? is reset after every command.** If you need to test the exit code of a command, check `$?` on the very next line, or save it: `rc=$?`. An intervening `echo` or assignment will overwrite it with its own exit code (usually 0).

2 — Strict Mode: `set -euo pipefail`

These three options — almost always combined — form the backbone of defensive scripting. Put them at the top of every non-trivial script.

The strict mode header

```
#!/bin/bash
set -euo pipefail

# Equivalent to writing all three separately:
# set -e          Exit immediately if any command fails
# set -u          Treat unset variables as an error
# set -o pipefail Make a pipeline fail if ANY stage fails
```

set -e — exit on error

What set -e does and when it doesn't trigger

```
set -e

# Without set -e: script continues after failure
ls /nonexistent   # exits 2 — script would continue silently!
echo "This runs"

# With set -e: script exits immediately on that ls failure

# set -e does NOT trigger for:
#   - commands in an if condition
#   - the left side of && or ||
#   - commands followed by ! (negation)
#   - commands in a while/until condition

if grep -q "pattern" file.txt; then   # grep's exit code is handled here
    echo "found"
fi

# Use || true to intentionally allow a command to fail
rm stale_lock.pid || true             # don't abort if file doesn't exist
mkdir -p output/ || true              # -p already handles this, but pattern is common
```

set -u — catch unset variables

set -u in action

```
set -u

# Without set -u: typos silently expand to empty string
username="alice"
echo "Hello, $usrname"               # typo — prints "Hello, " silently

# With set -u: bash throws an error immediately
bash: usrname: unbound variable

# Use ${var:-default} to safely allow a variable to be unset
log_level="${LOG_LEVEL:-info}"      # default to "info" if LOG_LEVEL not set
output_dir="${1:-/tmp/output}"     # default to /tmp/output if $1 not given

# Special variables $@ and $* need care with set -u
# Use "${@:-}" or check $# first
[[ $# -gt 0 ]] && echo "First arg: $1"
```

set -o pipefail — catch pipeline failures

Why pipefail matters

```
# Without pipefail: pipeline exit code = last command's code
# This silently succeeds even though cat failed!
cat /nonexistent/file.txt | grep "pattern"
echo "Exit: $?"                      → 1  (grep's code, not cat's)

set -o pipefail

cat /nonexistent/file.txt | grep "pattern"
cat: /nonexistent/file.txt: No such file or directory
# Pipeline exit code is now 1 (cat's failure), script exits

# PIPESTATUS — array of exit codes for each stage of last pipeline
cat file.txt | grep "x" | sort
echo "cat: ${PIPESTATUS[0]}, grep: ${PIPESTATUS[1]}, sort: ${PIPESTATUS[2]}"
```

3 — The trap Command

`trap` registers a command or function to run when the script receives a signal or exits. It is essential for cleanup — removing temp files, releasing locks, printing a useful error message — no matter how the script ends.

trap syntax and signal names

```
# trap 'command' SIGNAL [SIGNAL...]

# Key pseudo-signals:
#   EXIT   — runs when the script exits for any reason
#   ERR    — runs after any command that returns non-zero (with set -e)
#   INT    — Ctrl+C (SIGINT)
#   TERM   — kill command (SIGTERM)
#   HUP    — terminal hang-up (SIGHUP)
#   DEBUG  — runs before every command (useful for tracing)

# Remove a trap
trap - EXIT

# List current traps
trap -p
```

EXIT trap — guaranteed cleanup

Cleanup on exit with a temp directory

```
#!/bin/bash
set -euo pipefail

# Create a temp directory and guarantee its removal on exit
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

# Everything in TMPDIR is cleaned up whether script succeeds,
# fails, or is killed with Ctrl+C
echo "Working in: $TMPDIR"
cp important_data.csv "$TMPDIR/"
# ... do processing ...
mv "$TMPDIR/result.csv" ./final_result.csv

# Cleanup happens automatically at this point
```

ERR trap — error location reporting

ERR trap that prints which line failed

```
#!/bin/bash
set -euo pipefail

on_error() {
    local exit_code=$?
    local line_no="$1"
    printf '\n\033[31m[ERROR]\033[0m Script failed at line %d (exit code %d)\n' \
        "$line_no" "$exit_code" >&2
}

# $LINENO expands to the current line number at the time trap fires
trap 'on_error $LINENO' ERR

# Combine with EXIT for full coverage
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

echo "Script starting..."
cp /nonexistent "$TMPDIR"    # this will fail
echo "This line is never reached"

[ERROR] Script failed at line 18 (exit code 1)
```

INT and TERM — graceful interruption

Handling Ctrl+C and kill signals

```
#!/bin/bash
LOCK_FILE="/var/run/myscript.lock"

cleanup() {
    echo ""
    echo "Caught signal — cleaning up..." >&2
    rm -f "$LOCK_FILE"
    exit 130    # 128 + SIGINT(2) — conventional exit code for Ctrl+C
}

trap 'cleanup' INT TERM

# Acquire lock
touch "$LOCK_FILE"
echo "Running (PID $$). Press Ctrl+C to stop."

while true; do
    echo "Working..."
    sleep 2
done
```

4 — Error Handling Patterns

die() — centralised fatal error function

A reusable die() function

```
die() {
    local msg="${1:-Fatal error}"
    local code="${2:-1}"
    printf '\033[31m[FATAL]\033[0m %s\n' "$msg" >&2
    exit "$code"
}

# Usage — call it anywhere you want to abort with a message
[[ -f "$config" ]] || die "Config file not found: $config"

ping -c1 -W1 "$host" >/dev/null 2&1 || die "Host unreachable: $host"

[[ $EUID -eq 0 ]] || die "This script must be run as root" 2
```

require() — checking dependencies upfront

Verify required tools exist before doing any work

```
require() {
    local cmd
    for cmd in "$@"; do
        command -v "$cmd" >/dev/null 2&1 || \
            die "Required command not found: $cmd"
    done
}

# At the top of the script, check all dependencies at once
require curl jq awk sed git

# command -v is preferred over which for portability
# It returns 0 if found, non-zero if not
```

Handling errors in subshells and command substitution

set -e and command substitution gotchas

```
set -e

# GOTCHA: set -e is NOT inherited by command substitution $()
result=$(failing_command)     # failing_command runs in a subshell
# The assignment itself fails — set -e DOES catch this

# BUT if you assign in a local declaration, set -e is bypassed!
bad_example() {
    local val=$(failing_command)    # local always exits 0 — failure is hidden!
}

# CORRECT: declare local first, assign separately
good_example() {
    local val
    val=$(failing_command)          # now set -e sees the failure
}

# Explicitly check when you need the value AND the exit code
output=$(some_command) || { echo "some_command failed" >&2; exit 1; }
```

Critical: `local var=$(cmd)` silently swallows errors. The `local` builtin always exits with code 0, masking the failure of the command substitution inside it. Always declare `local var` on one line and assign it on the next. This is one of the most common silent failure bugs in Bash scripts.

5 — Structured Logging

Good logging is what tells you *what happened* after a script runs unattended. A minimal log library takes only a dozen lines to write and pays dividends immediately.

A reusable log library (lib/log.sh)

```
#!/bin/bash
# lib/log.sh — source this from your scripts

LOG_LEVEL="${LOG_LEVEL:-INFO}"     # override with: LOG_LEVEL=DEBUG ./script.sh
LOG_FILE="${LOG_FILE:-}"           # set to a path to also write to a file

declare -A _LOG_LEVELS=( [DEBUG]=0 [INFO]=1 [WARN]=2 [ERROR]=3 )

_log() {
    local level="$1"; shift
    local msg="$*"
    local ts
    ts=$(date '+%Y-%m-%d %H:%M:%S')

    # Skip if below configured log level
    [[ "${_LOG_LEVELS[$level]:-0}" -lt "${_LOG_LEVELS[$LOG_LEVEL]:-1}" ]] && return

    local colour
    case "$level" in
        DEBUG) colour='\033[36m' ;;   # cyan
        INFO)  colour='\033[32m' ;;   # green
        WARN)  colour='\033[33m' ;;   # yellow
        ERROR) colour='\033[31m' ;;   # red
    esac

    local line
    line="[$ts] [${level}] $msg"

    # Coloured output to stderr
    printf "${colour}%s\033[0m\n" "$line" >&2

    # Plain output to log file (no colour codes)
    [[ -n "$LOG_FILE" ]] && printf "%s\n" "$line" >> "$LOG_FILE"
}

log_debug() { _log DEBUG "$@"; }
log_info()  { _log INFO  "$@"; }
log_warn()  { _log WARN  "$@"; }
log_error() { _log ERROR "$@"; }
```

Using the log library in a script

```
#!/bin/bash
set -euo pipefail
source "$(dirname "$0")/lib/log.sh"

LOG_FILE="/var/log/myscript.log"

log_info  "Script started (PID $$)"
log_debug "Arguments: $*"

if ! ping -c1 -W1 google.com >/dev/null 2&1; then
    log_warn "No network connectivity"
fi

log_info "Processing complete"

# Run with debug logging:
#   LOG_LEVEL=DEBUG ./myscript.sh
[2026-06-09 14:32:01] [INFO] Script started (PID 4521)
[2026-06-09 14:32:01] [DEBUG] Arguments: file.csv
[2026-06-09 14:32:02] [INFO] Processing complete
```

6 — Debugging Tools

set -x — execution tracing

Trace mode: see every command as it executes

```
# Enable at the command line — no script modification needed
bash -x myscript.sh arg1 arg2

# Or add to the script header alongside other options
set -euxo pipefail

# Enable/disable around a specific section only
echo "Before the tricky bit"
set -x
complex_operation "$arg1" "$arg2"
set +x
echo "After the tricky bit"

# Customise the trace prompt — show line numbers
PS4='+ ${BASH_SOURCE[0]}:${LINENO}: '
set -x

# Trace output (each line prefixed with ++):
++ myscript.sh:12: cp source.txt /tmp/
++ myscript.sh:13: echo "Done"
```

bash -n — syntax check without running

Validate syntax before executing

```
# Check syntax of a script — runs no commands
bash -n myscript.sh
# No output = no syntax errors
# bash -n only catches syntax errors, NOT logic errors or missing files

# Combine with set -e in a CI pipeline:
# bash -n is fast — run it first before the real execution
```

ShellCheck — static analysis

ShellCheck is the single most valuable tool for Bash development. It performs static analysis and catches:

- Quoting bugs (`$var` instead of `"$var"`)
- The `local var=$(cmd)` silent failure pattern
- Unquoted globs and word-splitting issues
- Portability problems (bash-only features in `#!/bin/sh` scripts)

- Common logic errors and deprecated syntax

Install: `apt install shellcheck` / `brew install shellcheck`
Run: `shellcheck myscript.sh`
Online: **shellcheck.net** — paste your script for instant analysis.

Debugging techniques in practice

Other useful debugging patterns

```
# 1. Print variable contents and types
declare -p my_array             # shows type and value — great for arrays
declare -p my_var

# 2. Check where a function is defined
declare -f function_name        # prints the function body

# 3. Print a stack trace on error
print_stack() {
    local i=0
    echo "Call stack:" >&2
    while caller $i; do
        (( i++ ))
    done >&2
}
trap 'print_stack' ERR

# 4. Time a section of code
start=$(date +%s%N)             # nanoseconds
# ... work ...
elapsed=$(( ($(date +%s%N) - start) / 1000000 ))
echo "Elapsed: ${elapsed}ms"

# 5. BASH_SOURCE, FUNCNAME, LINENO — where am I?
debug_location() {
    printf "[%s:%d in %s()]\n" \
        "${BASH_SOURCE[1]}" "${BASH_LINENO[0]}" "${FUNCNAME[1]}" >&2
}

# 6. Pause and inspect mid-script
breakpoint() {
    set +x
    read -r -p "[breakpoint] Press Enter to continue..."
    set -x
}
```

7 — The Defensive Script Template

This is a production-ready starting point that combines everything from this chapter. Copy it as your base for any non-trivial script.

Complete defensive script skeleton

```bash
#!/usr/bin/env bash
# ============================================================
# script_name.sh — One-line description of what this does
# Usage: ./script_name.sh [OPTIONS] ARGUMENT
# ============================================================
set -euo pipefail
IFS=$'\n\t'    # word-split only on newlines and tabs, not spaces

# ── Script metadata ──────────────────────────────────────────
readonly SCRIPT_NAME="$(basename "${BASH_SOURCE[0]}")"
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# ── Logging ──────────────────────────────────────────────────
LOG_LEVEL="${LOG_LEVEL:-INFO}"
log()  { printf '[%s] [%s] %s\n' "$(date '+%H:%M:%S')" "$1" "$2" >&2; }
info() { log INFO  "$*"; }
warn() { log WARN  "$*"; }
die()  { log FATAL "$*"; exit 1; }

# ── Cleanup ──────────────────────────────────────────────────
TMPDIR=$(mktemp -d)

cleanup() {
    local rc=$?
    rm -rf "$TMPDIR"
    [[ $rc -ne 0 ]] && warn "Script exited with code $rc"
}
trap 'cleanup' EXIT

on_error() { die "Unexpected error on line $1"; }
trap 'on_error $LINENO' ERR

# ── Argument parsing ─────────────────────────────────────────
usage() {
    printf 'Usage: %s [--verbose] INPUT_FILE\n' "$SCRIPT_NAME"
    exit "${1:-0}"
}

verbose=0
input_file=""

while [[ $# -gt 0 ]]; do
    case "$1" in
        --verbose|-v) verbose=1; shift ;;
        --help|-h)    usage ;;
        --)           shift; break ;;
        -*)           die "Unknown option: $1" ;;
```

```
        *)          input_file="$1"; shift ;;
    esac
done

[[ -n "$input_file" ]] || usage 2
[[ -f "$input_file" ]] || die "File not found: $input_file"

# ── Main logic ──────────────────────────────────────────
main() {
    info "Starting $SCRIPT_NAME"
    # ... your work here ...
    info "Done"
}

main "$@"
```

❌ FRAGILE SCRIPT

```
#!/bin/bash
# No strict mode
# No trap
# No error checking

cp $1 /backup/
rm $1
echo "Done"
# If cp fails: rm runs anyway
# If $1 has spaces: breaks
# If /backup/ full: silent fail
```

✅ DEFENSIVE SCRIPT

```
#!/usr/bin/env bash
set -euo pipefail
trap 'echo "Failed on line $LINENO" >&2' E

[[ -f "${1:?No file given}" ]] \
    || { echo "Not a file: $1" >&2; exit 1

if cp "$1" /backup/; then
    rm "$1"
    echo "Done"
else
    echo "Copy failed — original kept" >&2
    exit 1
fi
```

8 — Quick Reference

TOOL / OPTION	WHAT IT DOES	NOTES
`$?`	Exit code of most recent command	0 = success; save to `rc=$?` before it's overwritten
`set -e`	Exit on any command failure	Does not trigger in `if`, `||`, `&&`, `!` contexts
`set -u`	Error on unset variable	Use `${var:-default}` for intentionally optional vars

TOOL / OPTION	WHAT IT DOES	NOTES
`set -o pipefail`	Pipeline fails if any stage fails	`PIPESTATUS` array has per-stage codes
`set -x`	Print each command before executing	Customise prompt with `PS4`
`bash -n script`	Syntax check without running	Quick pre-flight check
`trap 'cmd' EXIT`	Run on any exit	Use for cleanup — temp files, locks
`trap 'cmd' ERR`	Run after any error (with `set -e`)	Use `$LINENO` to report location
`trap 'cmd' INT TERM`	Handle Ctrl+C / kill	Exit with code 130 for INT
`trap - SIGNAL`	Remove a trap	—
`command -v name`	Check a command exists (portable)	Prefer over `which`
`declare -p var`	Print variable type and value	Essential for debugging arrays
`caller N`	Print call stack frame N	Use in a loop for full stack trace
`shellcheck`	Static analysis — catches subtle bugs	Run on every script before committing

🪥 Exercises

Apply what you have learned. Write each script yourself before looking at the sample solution.

EXERCISE 1

Add full defensive error handling to this broken script. It should: (1) use strict mode, (2) trap EXIT to remove a temp file it creates, (3) trap ERR to print the failing line number, (4) validate that exactly one argument was given and that it is a readable file, (5) check that `awk` is available before using it.

Broken script to fix:

```
awk '{sum+=$1} END{print sum}' $1 > /tmp/out.txt; echo "Total: $(cat /tmp/out.txt)"
```

Hint: wrap the logic in a `main()` function called at the end. Use `${1:?...}` for argument checking, `[[ -r "$1" ]]` to verify readability, and `command -v awk` to check availability. Create the temp file with `mktemp` and trap its removal on EXIT.

▶ Show Sample Solution

EXERCISE 2

Write a script called `safe_deploy.sh` that simulates a deployment with five steps (each represented by a function that may or may not succeed). Use a full defensive setup: strict mode, an ERR trap that logs the failing step name and line number, an EXIT trap that logs whether the deployment succeeded or failed (based on the exit code), and a `rollback()` function that is called on ERR to undo any completed steps. Each step should log its progress using a simple log function.

Hint: track completed steps in an array. In the rollback function, iterate the array in reverse and call an undo function for each step. The EXIT trap can read `$?` to determine success or failure. Simulate random step failure with `(( RANDOM % 3 == 0 ))`.

▶ Show Sample Solution

EXERCISE 3

Write a script called `retry.sh` that wraps any command and retries it up to N times with a configurable delay between attempts. Usage: `./retry.sh --attempts 5 --delay 2 -- curl https://example.com`. Log each attempt number, whether it succeeded or failed, and the exit code. Exit 0 only if the command eventually succeeds; exit 1 if all attempts fail.

Hint: parse `--attempts` and `--delay` from `$@`, stopping at `--`. Use a `for` loop with a C-style counter. Capture the command's exit code with `cmd_rc=$?` inside a subshell. Use `sleep "$delay"` between attempts and skip the sleep after the final attempt.

▶ Show Sample Solution

EXERCISE 4

Write a script called `health_check.sh` that checks a list of services and URLs. For each service (e.g. `nginx`, `ssh`), it should verify the service is running using `systemctl is-active`. For each URL, it should check HTTP reachability using `curl -sf`. Results should be logged with coloured PASS/FAIL labels. At the end, print a summary count of passes and failures. Exit 0 if all checks pass, exit 1 if any fail.

Hint: define arrays for services and URLs at the top of the script. Use a generic `check()` function that takes a label and a command; it runs the command, captures the exit code, and prints PASS in green or FAIL in red using `printf '\033[32mPASS\033[0m'`. Count failures in a variable, not with `set -e`.

▶ Show Sample Solution

CHAPTER 12 OF 12

Topic 12 — Practical Script Design

📐 Topic 12 — Practical Script Design

The previous eleven chapters gave you the language. This final chapter is about the craft — the decisions and patterns that separate a script you wrote once and never want to touch again from one you are confident running in production. We cover argument parsing, configuration management, idempotency, locking, output design, modular organisation, and testing. The chapter closes with a complete, fully-annotated real-world script that draws on every major topic in the course.

1 — Argument Parsing

Scripts that go beyond a single required argument need proper option parsing. There are two good approaches: the built-in `getopts` (short flags only) and a manual `while/case` loop (short and long flags).

getopts — built-in short option parser

getopts — handles -v, -o file, -vn etc.

```
#!/usr/bin/env bash
set -euo pipefail

verbose=0
output="output.txt"
dry_run=0

usage() {
    cat <<EOF
Usage: $(basename "$0") [OPTIONS] INPUT_FILE

Options:
  -v          Enable verbose output
  -o FILE     Write output to FILE (default: output.txt)
  -n          Dry run — show what would happen without doing it
  -h          Show this help
EOF
    exit "${1:-0}"
}

# getopts string: each letter is a flag; a colon after means it takes an argument
while getopts "vno:h" opt; do
    case "$opt" in
        v) verbose=1 ;;
        n) dry_run=1 ;;
        o) output="$OPTARG" ;;
        h) usage ;;
        *) usage 2 ;;
    esac
done
shift $(( OPTIND - 1 ))    # remove parsed options; $1 is now the first positional arg

[[ $# -ge 1 ]] || usage 2
input_file="$1"

[[ $verbose -eq 1 ]] && echo "Verbose mode on. Output: $output"
```

`getopts` handles combined flags (`-vn`), option arguments with or without a space (`-o file` or `-ofile`), and `--` to end option parsing. It does NOT support long options like `--verbose`.

Manual while/case — short and long options

Manual parsing — supports --verbose, --output=FILE, etc.

```
verbose=0; dry_run=0; output="output.txt"

while [[ $# -gt 0 ]]; do
    case "$1" in
        -v|--verbose)
            verbose=1; shift ;;
        -n|--dry-run)
            dry_run=1; shift ;;
        -o|--output)
            output="${2:?--output requires a value}"; shift 2 ;;
        --output=*)
            output="${1#--output=}"; shift ;;      # --output=value form
        -h|--help)
            usage; exit 0 ;;
        --)
            shift; break ;;                          # end of options
        -*)
            echo "Unknown option: $1" >&2; exit 2 ;;
        *)
            break ;;                                  # first non-option arg
    esac
done
# Remaining positional arguments are in "$@"
```

2 — Configuration Management

Well-designed scripts read configuration from multiple sources in a defined precedence order: built-in defaults are overridden by a config file, which is overridden by environment variables, which are overridden by command-line flags. This makes scripts flexible without being fragile.

Configuration precedence pattern

```
#!/usr/bin/env bash

# ── 1. Hard-coded defaults ──────────────────────────────
DB_HOST="localhost"
DB_PORT="5432"
DB_NAME="myapp"
LOG_LEVEL="INFO"
BACKUP_DIR="/var/backups/myapp"

# ── 2. Load config file (if it exists) ──────────────────
CONFIG_FILE="${CONFIG_FILE:-/etc/myapp/myapp.conf}"
if [[ -f "$CONFIG_FILE" ]]; then
    # shellcheck source=/dev/null
    source "$CONFIG_FILE"
fi

# ── 3. Environment variables override config file ───────
# (Already set in environment — no action needed if we used
#  the same variable names, since sourcing the config file
#  would override env vars. Use a different naming convention:)
DB_HOST="${MYAPP_DB_HOST:-$DB_HOST}"
DB_PORT="${MYAPP_DB_PORT:-$DB_PORT}"
LOG_LEVEL="${MYAPP_LOG_LEVEL:-$LOG_LEVEL}"

# ── 4. Command-line flags override everything (parsed earlier) ─
# (already set by getopts/while-case above)

# ── Validate required configuration ─────────────────────
[[ -n "$DB_HOST" ]] || die "DB_HOST is not set"
[[ "$DB_PORT" =~ ^[0-9]+$ ]] || die "DB_PORT must be numeric: $DB_PORT"
```

> **Convention:** *use `APPNAME_VARNAME` for environment variables (e.g. `MYAPP_DB_HOST`) to avoid clashing with system variables. Inside the script, use shorter local names. Document the full list of supported environment variables in the `--help` output.*

3 — Idempotency

An **idempotent** script produces the same result whether it has been run once or ten times. This is essential for deployment scripts, cron jobs, and anything that might be retried after failure. The golden rule: *check before you act*.

Idempotent patterns

```
# ── Creating files and directories ──────────────────────
mkdir -p /etc/myapp/conf.d              # -p: no error if already exists

[[ -f /etc/myapp/default.conf ]] \
    || cp default.conf /etc/myapp/          # only copy if not there yet

# ── Installing packages ─────────────────────────────────
# Bad: always runs dpkg
apt-get install -y nginx

# Better: skip if already installed
dpkg -s nginx >/dev/null 2&1 || apt-get install -y nginx

# ── Adding a line to a file (only once) ─────────────────
line="export PATH=\$PATH:/opt/myapp/bin"
grep -qxF "$line" ~/.bashrc || echo "$line" >> ~/.bashrc

# ── Creating a symlink ──────────────────────────────────
ln -sf /opt/myapp/bin/myapp /usr/local/bin/myapp    # -f: replace if exists

# ── Conditional database migration ──────────────────────
schema_version=$(psql -tAc "SELECT version FROM schema_migrations ORDER BY id DESC LIMIT 1
if [[ "$schema_version" -lt 42 ]]; then
    psql -f migration_042.sql
fi
```

4 — Script Locking

When a script must not run concurrently with itself — a backup job, a queue processor, a cron task — use a lock file. The safest implementation uses `flock`, which is atomic and automatically releases the lock if the process dies.

flock — advisory locking

```
#!/usr/bin/env bash
set -euo pipefail

LOCK_FILE="/var/run/myapp.lock"

# Method 1: flock wraps the entire script (simplest)
# Re-execute the script under flock if not already locked
[ "${FLOCKER:-}" != "$0" ] && \
    exec env FLOCKER="$0" flock -en "$LOCK_FILE" "$0" "$@" || \
    { echo "Already running — exiting" >&2; exit 1; }

# Method 2: open a file descriptor to the lock file
exec 200<>"$LOCK_FILE"           # open fd 200 for read+write
flock -n 200 || {
    echo "Another instance is running (PID: $(cat "$LOCK_FILE"))" >&2
    exit 1
}
# Write our PID to the lock file so others can identify us
echo $$ >&200

# Lock is released automatically when fd 200 closes at script exit
# No explicit unlock needed — even if the script crashes

# Portable fallback (no flock): mkdir is atomic on most filesystems
LOCK_DIR="/tmp/myapp.lock"
if ! mkdir "$LOCK_DIR" 2>/dev/null; then
    echo "Script already running" >&2; exit 1
fi
trap 'rmdir "$LOCK_DIR"' EXIT
```

5 — Output Design

Well-designed output makes scripts easy to use interactively and easy to parse in automation. The key principles: write progress/status to **stderr**, write data to **stdout**; detect whether output is a terminal before adding colour; and give users a `--quiet` mode when the script is used in pipelines.

Detecting terminal and colour support

Colour output that degrades gracefully

```
# Only use colours when stderr is a real terminal
if [[ -t 2 ]]; then
    RED='\033[0;31m'
    GREEN='\033[0;32m'
    YELLOW='\033[0;33m'
    BLUE='\033[0;34m'
    BOLD='\033[1m'
    RESET='\033[0m'
else
    RED=""; GREEN=""; YELLOW=""; BLUE=""; BOLD=""; RESET=""
fi

# -t N: true if file descriptor N is open and is a terminal
# -t 1 → stdout is a terminal
# -t 2 → stderr is a terminal

info()    { printf "${GREEN}✓${RESET} %s\n" "$*" >&2; }
warn()    { printf "${YELLOW}⚠${RESET} %s\n" "$*" >&2; }
error()   { printf "${RED}✗${RESET} %s\n" "$*" >&2; }
heading() { printf "\n${BOLD}%s${RESET}\n" "$*" >&2; }
```

A simple spinner for long-running tasks

Spinner that runs while a background job works

```
spinner() {
    local pid=$1
    local msg="${2:-Working...}"
    local frames=( '⠋' '⠙' '⠹' '⠸' '⠼' '⠴' '⠦' '⠧' '⠇' '⠏' )
    local i=0
    while kill -0 "$pid" 2>/dev/null; do
        printf "\r  %s %s" "${frames[$i]}" "$msg" >&2
        i=$(( (i + 1) % ${#frames[@]} ))
        sleep 0.1
    done
    printf "\r  \033[32m✓\033[0m %s\n" "$msg" >&2
}

# Usage: run something in the background, spin while it works
# heavy_command arg1 arg2 &
# spinner $! "Compressing archive..."
# wait $!   # pick up its exit code

example_usage() {
    sleep 3 &                          # simulate a long operation
    spinner $! "Backing up database..."
    wait $!
}
```

Prompting for confirmation

Confirmation prompts and non-interactive mode

```
force=0   # set with --force / -f flag

confirm() {
    local msg="${1:-Are you sure?}"
    # Skip prompt in non-interactive mode or when --force is set
    [[ $force -eq 1 ]] && return 0
    [[ ! -t 0 ]] && { echo "Non-interactive mode — use --force to proceed" >&2; return 1;
    read -r -p "${msg} [y/N] " reply
    [[ "$reply" == [yY] ]]
}

if confirm "Delete all logs in /var/log/myapp?"; then
    rm -rf /var/log/myapp/*.log
    info "Logs deleted"
else
    info "Aborted"
fi
```

6 — Modular Organisation

Once a collection of scripts shares common functions — logging, config loading, output helpers — extract them into library files and `source` them. This eliminates copy-paste drift and makes the shared code testable in isolation.

Recommended project layout

```
myapp/
├── bin/
│   ├── backup.sh          # entry-point scripts
│   ├── deploy.sh
│   └── health_check.sh
├── lib/
│   ├── log.sh             # shared libraries
│   ├── config.sh
│   └── utils.sh
├── tests/
│   ├── test_utils.bats    # BATS test files
│   └── test_config.bats
└── myapp.conf.example
```

Locating and sourcing library files reliably

```
#!/usr/bin/env bash
# bin/deploy.sh

# Find the script's own directory regardless of how it was invoked
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
readonly LIB_DIR="${SCRIPT_DIR}/../lib"

# Source libraries — use a guard so they can be sourced multiple times safely
source "${LIB_DIR}/log.sh"
source "${LIB_DIR}/config.sh"
source "${LIB_DIR}/utils.sh"

# The guard pattern inside each library file:
#   [[ -n "${_LOG_LOADED:-}" ]] && return
#   readonly _LOG_LOADED=1
#   ... function definitions ...
```

7 — Testing Bash Scripts

The best tool for testing Bash is **BATS** (Bash Automated Testing System). Tests are written as plain Bash with a thin assertion layer on top. Even without BATS, well-structured scripts can be tested with a few conventions.

BATS test structure

```
#!/usr/bin/env bats
# tests/test_utils.bats
# Install: npm install -g bats  OR  apt install bats
# Run:     bats tests/

# Load the library to test
setup() {
    source "${BATS_TEST_DIRNAME}/../lib/utils.sh"
}

# Each @test block is one test case
@test "is_integer: accepts valid integers" {
    run bash -c 'source lib/utils.sh; is_integer 42 && echo yes'
    [ "$status" -eq 0 ]
    [ "$output" = "yes" ]
}

@test "is_integer: rejects strings" {
    run bash -c 'source lib/utils.sh; is_integer "hello"'
    [ "$status" -eq 1 ]
}

@test "slugify: converts spaces to hyphens" {
    run bash -c 'source lib/utils.sh; slugify "Hello World"'
    [ "$output" = "hello-world" ]
}

@test "backup creates output file" {
    local tmpdir
    tmpdir=$(mktemp -d)
    run ./bin/backup.sh --output "$tmpdir" ./fixtures/sample.txt
    [ "$status" -eq 0 ]
    [ -f "${tmpdir}/sample.txt.bak" ]
    rm -rf "$tmpdir"
}
```

> **Design for testability:** *keep logic in functions, not at the top level. Use a `main()` function called at the very bottom of the script. This lets you `source` the script in a test to load the functions without executing them, exactly as you would with any library file.*

8 — A Complete Real-World Script

This script ties together all twelve topics. It performs a configurable, logged, idempotent database backup with rotation — the kind of thing you would actually schedule in cron.

db_backup.sh ~180 lines · strict mode · trap · logging · config · locking · idempotency · rotation

```
#!/usr/bin/env bash
# ============================================================
# db_backup.sh — PostgreSQL database backup with rotation
# Usage: ./db_backup.sh [OPTIONS]
#
# Environment variables (override config file):
#   BACKUP_DB_HOST    DB_USER    DB_NAME    BACKUP_DIR
#   BACKUP_KEEP_DAYS  LOG_LEVEL  LOG_FILE
# ============================================================
set -euo pipefail
IFS=$'\n\t'

# ── Script metadata ──────────────────────────────────────────
readonly SCRIPT_NAME="$(basename "${BASH_SOURCE[0]}")"
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
readonly TIMESTAMP="$(date '+%Y%m%d_%H%M%S')"

# ── Defaults ─────────────────────────────────────────────────
DB_HOST="localhost"
DB_PORT="5432"
DB_USER="postgres"
DB_NAME="myapp"
BACKUP_DIR="/var/backups/db"
KEEP_DAYS="7"
LOG_LEVEL="INFO"
LOG_FILE=""
COMPRESS="1"
LOCK_FILE="/tmp/${SCRIPT_NAME}.lock"

# ── Colour setup ─────────────────────────────────────────────
if [[ -t 2 ]]; then
    R='\033[31m' G='\033[32m' Y='\033[33m' B='\033[1m' X='\033[0m'
else
    R="" G="" Y="" B="" X=""
fi

# ── Logging ──────────────────────────────────────────────────
declare -A _LL=( [DEBUG]=0 [INFO]=1 [WARN]=2 [ERROR]=3 )

_log() {
    local lvl="$1"; shift
    [[ "${_LL[$lvl]:-0}" -lt "${_LL[$LOG_LEVEL]:-1}" ]] && return
    local ts="$(date '+%H:%M:%S')"
    local col
    case "$lvl" in
        DEBUG) col='\033[36m' ;; INFO)  col="$G" ;;
        WARN)  col="$Y"       ;; ERROR) col="$R" ;;
    esac
    printf "${col}[%s][%s]${X} %s\n" "$ts" "$lvl" "$*" >&2
```

```
    [[ -n "$LOG_FILE" ]] && \
        printf "[%s][%s] %s\n" "$ts" "$lvl" "$*" >> "$LOG_FILE"
}
log_debug() { _log DEBUG "$@"; }
log_info()  { _log INFO  "$@"; }
log_warn()  { _log WARN  "$@"; }
log_error() { _log ERROR "$@"; }
die()       { log_error "$*"; exit 1; }

# ── Cleanup / trap ──────────────────────────────────────────
TMPDIR=""

cleanup() {
    local rc=$?
    [[ -n "$TMPDIR" ]] && rm -rf "$TMPDIR"
    [[ $rc -ne 0 ]] && log_error "Script exited with code $rc"
}
trap 'cleanup' EXIT
trap 'die "Unexpected error on line $LINENO"' ERR

# ── Usage ───────────────────────────────────────────────────
usage() {
    cat <<EOF
Usage: $SCRIPT_NAME [OPTIONS]

  -H HOST   Database host (default: $DB_HOST)
  -p PORT   Database port (default: $DB_PORT)
  -u USER   Database user (default: $DB_USER)
  -d DB     Database name (default: $DB_NAME)
  -o DIR    Backup output directory (default: $BACKUP_DIR)
  -k DAYS   Keep backups for N days (default: $KEEP_DAYS)
  -n        No compression
  -v        Verbose (DEBUG) logging
  -h        Show this help
EOF
    exit "${1:-0}"
}

# ── Argument parsing ────────────────────────────────────────
while getopts "H:p:u:d:o:k:nvh" opt; do
    case "$opt" in
        H) DB_HOST="$OPTARG"   ;; p) DB_PORT="$OPTARG"   ;;
        u) DB_USER="$OPTARG"   ;; d) DB_NAME="$OPTARG"   ;;
        o) BACKUP_DIR="$OPTARG" ;; k) KEEP_DAYS="$OPTARG" ;;
        n) COMPRESS="0"        ;; v) LOG_LEVEL="DEBUG"   ;;
        h) usage               ;; *) usage 2             ;;
    esac
done
```

```
# ── Apply env var overrides (higher precedence than defaults) ──
DB_HOST="${BACKUP_DB_HOST:-$DB_HOST}"
DB_USER="${BACKUP_DB_USER:-$DB_USER}"
DB_NAME="${BACKUP_DB_NAME:-$DB_NAME}"

# ── Require pg_dump ──────────────────────────────────────────
command -v pg_dump >/dev/null 2&1 || die "pg_dump not found — install postgresql-client"

# ── Locking ──────────────────────────────────────────────────
exec 200<>"$LOCK_FILE"
flock -n 200 || die "Another backup is already running (lock: $LOCK_FILE)"
echo $$ >&200

# ── Main backup function ─────────────────────────────────────
do_backup() {
    log_info "Starting backup of ${DB_NAME} @ ${DB_HOST}:${DB_PORT}"

    # Idempotent: create output directory if it doesn't exist
    mkdir -p "$BACKUP_DIR"

    TMPDIR=$(mktemp -d)
    local dump_file="${TMPDIR}/${DB_NAME}_${TIMESTAMP}.sql"
    local final_file="${BACKUP_DIR}/${DB_NAME}_${TIMESTAMP}.sql"

    # Run pg_dump — pass password via .pgpass or PGPASSWORD env var
    log_debug "Running pg_dump to $dump_file"
    pg_dump -h "$DB_HOST" -p "$DB_PORT" -U "$DB_USER" \
            -Fp --no-password "$DB_NAME" > "$dump_file"

    # Optionally compress
    if [[ $COMPRESS -eq 1 ]]; then
        log_debug "Compressing..."
        gzip "$dump_file"
        dump_file="${dump_file}.gz"
        final_file="${final_file}.gz"
    fi

    # Atomic move to final location
    mv "$dump_file" "$final_file"

    local size
    size=$(du -sh "$final_file" | cut -f1)
    log_info "Backup saved: $final_file ($size)"
}

# ── Rotation function ────────────────────────────────────────
rotate_backups() {
    log_info "Removing backups older than ${KEEP_DAYS} days..."
    local count=0
```

```
    while IFS= read -r -d '' f; do
        log_debug "Removing: $f"
        rm "$f"
        (( count++ ))
    done <(find "$BACKUP_DIR" -name "${DB_NAME}_*.sql*" \
                  -type f -mtime +"${KEEP_DAYS}" -print0)
    [[ $count -gt 0 ]] && log_info "Removed $count old backup(s)"
    [[ $count -eq 0 ]] && log_debug "No old backups to remove"
}

# ── Entry point ──────────────────────────────────────────
main() {
    log_info "=== $SCRIPT_NAME started ==="
    do_backup
    rotate_backups
    log_info "=== $SCRIPT_NAME finished ==="
}

main "$@"
```

9 — The Ten Commandments of Bash Scripting

I **Always start with strict mode.** Every non-trivial script begins with `set -euo pipefail` and `IFS=$'\n\t'`. Silent failures are the most dangerous bugs.

II **Quote all variable expansions.** Write `"$var"` and `"${array[@]}"` everywhere. Unquoted expansions break on spaces and trigger unexpected glob expansion.

III **Use `[[ ]]`, not `[ ]`.** Double brackets handle empty variables gracefully, support `=~` regex matching, and never word-split or pathname-expand their operands.

IV **Trap EXIT for cleanup.** Create temp files with `mktemp` and register their removal immediately: `trap 'rm -rf "$TMPDIR"' EXIT`. Never rely on reaching the end of the script.

V **Never do `local var=$(cmd)`.** The `local` builtin masks the exit code of the substitution. Declare `local var` first, then assign `var=$(cmd)` on the next line.

VI **Write to stderr, pipe data through stdout.** Log messages, progress, and errors all go to `&2`. Only actual output data goes to stdout — so your script can be used in pipelines.

VII **Validate inputs at the top.** Check all arguments, files, and dependencies with `command -v`, `[[ -f ]]`, and regex validation before doing any real work.

VIII **Design for idempotency.** Ask: "what happens if this runs twice?" Use `mkdir -p`, `ln -sf`, `grep -qxF … || echo …`. A re-run should be safe.

IX **Run ShellCheck before committing.** It catches quoting bugs, masked failures, portability issues, and dozens of subtle traps that even experienced scripters miss. Make it part of your CI pipeline.

X **Wrap logic in functions, call `main "$@"` at the bottom.** This makes scripts sourceable, testable, and readable. The top level should contain only declarations and a single call to `main`.

10 — Quick Reference

PATTERN / TOOL	WHAT IT'S FOR	NOTES
`getopts "vo:h" opt`	Short option parsing	After loop: `shift $((OPTIND-1))`
`while/case "$1"`	Long + short option parsing	Handle `--opt=val` with `${1#--opt=}`
`${VAR:-default}`	Config precedence — fall through to default	Chain: CLI → env var → config file → default
`[[ -f "$f" ]] || cmd`	Idempotent file creation guard	Do the action only if the outcome isn't already there
`grep -qxF "line" file || echo "line" >> file`	Idempotent line append	`-x` whole line, `-F` literal, `-q` silent
`flock -n 200`	Prevent concurrent runs	Released automatically when the process ends
`mkdir "$LOCK_DIR" 2>/dev/null`	Portable atomic lock (no flock)	Trap `rmdir "$LOCK_DIR"` on EXIT
`[[ -t 2 ]]`	Test if stderr is a terminal	Use to suppress colour codes in scripts/pipes
`SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)`	Reliable self-location	Works regardless of how the script is called
`readonly VAR=val`	Prevent accidental overwrite	Good for SCRIPT_NAME, SCRIPT_DIR, TIMESTAMP
`bats tests/`	Run BATS test suite	Install: `apt install bats` / `brew install bats-core`

PATTERN / TOOL	WHAT IT'S FOR	NOTES
`shellcheck script.sh`	Static analysis	Non-negotiable — run on every script

🪛 Exercises

These final exercises ask you to design complete, production-quality scripts. Each one deliberately spans multiple topics from the course.

EXERCISE 1

Write a script called `setup_project.sh` that bootstraps a new project directory. It should accept a project name as an argument (validated as lowercase letters, digits, and hyphens only), create a standard directory structure (`src/`, `tests/`, `docs/`, `scripts/`), generate a `.gitignore`, a `README.md` with the project name, and an initial `scripts/run.sh`. The script must be fully idempotent — running it twice in the same directory should not overwrite existing files or produce errors.

Hint: validate the project name with `[[ =~ ^[a-z][a-z0-9-]+$ ]]`. Use `mkdir -p` for directories. For files, write a helper: `create_file_if_missing() { [[ -f "$1" ]] && return; cat > "$1" <<'EOF' ... EOF }`. Add strict mode, trap, and a `main()` function.

▶ Show Sample Solution

EXERCISE 2

Write a script called `monitor.sh` that runs continuously, checks disk usage on a configurable mount point every N seconds, and sends an alert (prints a coloured warning to stderr and appends to a log file) when usage exceeds a configurable threshold percentage. Support `--mount`, `--threshold`, `--interval`, and `--log-file` options. The script should handle Ctrl+C cleanly (print a summary of how many checks were run and how many alerts were triggered), and must not run two instances simultaneously.

Hint: parse disk usage with `df --output=pcent MOUNT | tail -1 | tr -d ' %'`. Store check/alert counts in variables incremented inside a `while true; do ... sleep "$interval"; done` loop. Use `trap 'print_summary; exit 0' INT TERM`. Use `flock` or a lock directory to prevent concurrent runs.

▶ Show Sample Solution

EXERCISE 3 — CAPSTONE

Write a script called `release.sh` that automates a software release process. It should: (1) accept a version string as an argument, validated as `vMAJOR.MINOR.PATCH` (e.g. `v1.4.2`); (2) check that the git working tree is clean; (3) run tests (simulate with a function that may pass or fail); (4) bump the version number in a `version.txt` file; (5) create a git tag; (6) build a release archive (`tar.gz` of the `src/` directory); (7) log every step with timestamps; and (8) support a `--dry-run` mode that shows exactly what would happen without making any changes.

Hint: create a `run_step()` function that takes a description and a command. In dry-run mode it prints the command prefixed with `[DRY-RUN]` instead of running it. Use `git status --porcelain` to check for uncommitted changes. Use `git tag -a "$version" -m "Release $version"` for tagging.

▶ Show Sample Solution