Bash Scripting — Intermediate Course

12-chapter intermediate-level course

CONTENTS

⚙ Intermediate Topic 1 — Advanced Parameter Expansion

The beginner course introduced the essentials of parameter expansion — defaults, string length, substring removal, and basic substitution. This chapter goes much deeper. Bash's parameter expansion system is one of the most powerful features of the shell, capable of replacing entire awk and sed pipelines with a single expression. We cover every operator in the language, the bash 4.4+ transformation flags, indirect expansion, namerefs, and compound patterns that combine multiple expansions into concise, readable one-liners.

1 — Default, Assign, Error, and Alternate Operators

You know `${var:-default}`. But there are four operators in this family, each with a *colon* and a *no-colon* variant — eight forms in total — and the distinction between them matters.

FORM	MEANING	TRIGGERS WHEN VAR IS…
`${var:-val}`	Use *val* if unset **or empty**	unset or `""`
`${var-val}`	Use *val* only if unset	unset only (empty string is fine)
`${var:=val}`	Assign *val* if unset or empty, then expand	unset or `""`
`${var=val}`	Assign *val* only if unset, then expand	unset only
`${var:?msg}`	Error and exit if unset or empty	unset or `""`
`${var?msg}`	Error and exit if unset	unset only
`${var:+val}`	Use *val* if set and non-empty (opposite of `:-`)	when var *has* a value
`${var+val}`	Use *val* if set at all (even if empty)	when var exists at all

The colon makes all the difference

```
empty=""
unset gone

# :- vs - (empty string triggers :- but NOT -)
echo "${empty:-fallback}"      → fallback    (empty triggers it)
echo "${empty-fallback}"       →              (empty string returned — no trigger)
echo "${gone:-fallback}"        → fallback
echo "${gone-fallback}"         → fallback

# := assigns as a side-effect — useful for lazy initialisation
```

```
echo "${cache:=computed_value}" → computed_value  (also sets $cache)
echo "$cache"                   → computed_value  (now set)
# Cannot use := on positional parameters ($1, $2) or special vars

# :? aborts with a message — great for required arguments
db_host="${DB_HOST:?DB_HOST environment variable must be set}"
# If DB_HOST is unset:  bash: DB_HOST: DB_HOST environment variable must be set

# :+ is the logical opposite of :- — "use this value IF the var IS set"
debug="1"
echo "${debug:+--verbose}"      → --verbose   (because debug is set)
echo "${unset_var:+--verbose}"  →              (nothing, because it's unset)

# Practical: conditionally add a flag to a command
curl "${verbose:+--silent}" https://example.com
# Passes --silent only when $verbose is empty/unset (backwards logic example)

# +  without colon: even an empty string counts as "set"
empty=""
echo "${empty+was_set}"         → was_set     (empty var still "exists")
echo "${gone+was_set}"          →              (unset var — nothing)
```

*The **colon variants** (:- := :? :+) treat an empty string the same as unset. The **no-colon variants** (- = ? +) only act when the variable is completely unset. In practice you almost always want the colon forms.*

2 — Substring Extraction and Prefix/Suffix Removal

Substring extraction — ${var:offset:length}

Slicing strings with offset and length

```
str="Hello, World!"

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

# ${var:offset:length}
echo "${str:7:5}"         → World
echo "${str:0:5}"         → Hello

# Negative offset counts from the END — note the space before the minus
# (space required to distinguish from :- default operator)
echo "${str: -6}"         → orld!
```

```
echo "${str: -6:5}"       → orld!  wait — 5 chars from position -6
echo "${str: -6:4}"       → orld

# Negative length means "exclude this many chars from the end"
echo "${str:7: -1}"       → World  (6 chars: "World!" minus last 1)

# Works on arrays too — returns a slice of elements
arr=( a b c d e f )
echo "${arr[@]:2:3}"      → c d e
echo "${arr[@]: -2}"      → e f

# Works on positional parameters
# Inside a function: ${@:2} = all args from the second onwards
all_but_first() { echo "${@:2}"; }
all_but_first one two three four
→ two three four
```

Pattern-based prefix and suffix removal — going deeper

 # ## % %% — with full glob patterns

```
path="/usr/local/lib/libmyapp.so.2.1.0"

# # removes shortest prefix matching the pattern
echo "${path#*/}"         → usr/local/lib/libmyapp.so.2.1.0

# ## removes longest prefix matching the pattern
echo "${path##*/}"        → libmyapp.so.2.1.0   (basename)

# % removes shortest suffix
echo "${path%/*}"         → /usr/local/lib      (dirname)
echo "${path%.*}"         → /usr/local/lib/libmyapp.so.2.1

# %% removes longest suffix
echo "${path%%.*}"        → /usr/local/lib/libmyapp

# Practical: strip all extensions from a filename
f="archive.tar.gz"
echo "${f%%.*}"           → archive

# Replace extension: strip suffix then add new one
echo "${f%.gz}.bz2"       → archive.tar.bz2

# Works in a loop — bulk rename files
```

```
for f in *.jpeg; do
    mv "$f" "${f%.jpeg}.jpg"
done

# Combined: extract just the base filename without any extension
file="/var/log/myapp/error.log.1"
base="${file##*/}"        # → error.log.1  (strip directory)
name="${base%%.*}"        # → error        (strip all extensions)
dir="${file%/*}"          # → /var/log/myapp  (dirname)
```

Search and replace — all four forms

/ // /# /% — replace first, all, prefix, suffix

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

# / — replace FIRST occurrence
echo "${str/the/a}"        → a cat sat on the mat

# // — replace ALL occurrences
echo "${str//the/a}"       → a cat sat on a mat

# /# — replace only if pattern matches at the START
echo "${str/#the/a}"       → a cat sat on the mat
echo "${str/#cat/a}"       → the cat sat on the mat  (no match — not at start)

# /% — replace only if pattern matches at the END
echo "${str/%mat/rug}"     → the cat sat on the rug

# Replace with empty string = deletion
echo "${str// /}"          → thecatsatonthemat  (remove all spaces)
echo "${str//[aeiou]/}"    → th ct st n th mt    (remove vowels)

# Patterns support globs
csv="one,  two ,  three  "
echo "${csv// /}"          → one,two,three  (remove all spaces)

# Works on each element of an array (expands to modified array)
names=( "Alice Smith" "Bob Jones" "Carol White" )
echo "${names[@]// /_}"    → Alice_Smith Bob_Jones Carol_White
```

3 — Case Conversion Operators

Bash 4.0 added four case-conversion operators. They're useful for normalising user input, building slugs, and formatting output without spawning `tr` or `awk`.

^ ^^ , ,, ~ ~~ — case conversion

```
str="Hello World"

# ^ — uppercase first character only
lower="hello world"
echo "${lower^}"          → Hello world

# ^^ — uppercase ALL characters
echo "${lower^^}"         → HELLO WORLD

# , — lowercase first character only
echo "${str,}"            → hello World

# ,, — lowercase ALL characters
echo "${str,,}"           → hello world

# ~ — toggle case of first character
echo "${str~}"            → hello World

# ~~ — toggle case of ALL characters
echo "${str~~}"           → hELLO wORLD

# All operators accept a glob pattern — only matching chars are converted
echo "${lower^^[aeiou]}"   → hEllO wOrld   (only vowels uppercased)
echo "${str,,[A-Z]}"      → hello world   (only uppercase letters lowercased)

# Practical: normalise input for case-insensitive comparison
read -r -p "Continue? [y/n]: " ans
if [[ "${ans,,}" == "y" || "${ans,,}" == "yes" ]]; then
    echo "Proceeding..."
fi

# Build a slug from a title
title="My Blog Post Title"
slug="${title,,}"         # → "my blog post title"
slug="${slug// /-}"       # → "my-blog-post-title"
```

The case operators require bash 4.0+. On macOS, the system `/bin/bash` may still be version 3.2 (due to GPL licensing). If portability to macOS system bash matters, use `tr` instead, or ensure a newer bash is installed via Homebrew.

4 — Indirect Expansion: `${!var}`

The `!` prefix makes bash expand the *value* of a variable as the *name* of another variable to expand. This allows dynamic variable lookup and is the classic way to implement dispatch tables without associative arrays.

Variable indirection — ${!varname}

```
fruit="apple"
apple="green"

# Normal expansion
echo "$fruit"              → apple

# Indirect: expand $fruit to get the name "apple", then expand $apple
echo "${!fruit}"           → green

# Indirect expansion with a variable holding a variable name
varname="PATH"
echo "${!varname}"         → /usr/local/bin:/usr/bin:/bin:...

# Practical: environment-specific config lookup
env="prod"
prod_host="db.example.com"
staging_host="db-staging.example.com"
dev_host="localhost"

host_var="${env}_host"     # → "prod_host"
echo "${!host_var}"        → db.example.com

# Dispatch table: map command names to function names
cmd_start="do_start"
cmd_stop="do_stop"
cmd_status="do_status"

do_start()  { echo "Starting..."; }
do_stop()   { echo "Stopping..."; }
do_status() { echo "Running."; }

action="start"
```

```
fn_var="cmd_${action}"      # → "cmd_start"
"${!fn_var}"                # calls do_start
Starting...
```

Listing variables by prefix — ${!prefix*} and ${!prefix@}

Enumerate all variables matching a prefix

```
DB_HOST="localhost"
DB_PORT="5432"
DB_NAME="myapp"
APP_ENV="production"

# ${!prefix*} expands to all variable NAMES starting with prefix
echo "${!DB_*}"             → DB_HOST DB_NAME DB_PORT

# Iterate over all DB_ variables and print their names + values
for varname in "${!DB_@}"; do
    printf "%-15s = %s\n" "$varname" "${!varname}"
done
DB_HOST         = localhost
DB_NAME         = myapp
DB_PORT         = 5432

# Useful for printing a config dump at script startup
dump_config() {
    echo "Configuration:"
    local v
    for v in "${!APP_@}" "${!DB_@}"; do
        printf "  %s=%s\n" "$v" "${!v}"
    done
}

# ${!prefix*} vs ${!prefix@}
# When unquoted: identical
# When quoted: * joins with IFS; @ produces separate words (like $* vs $@)
```

5 — Namerefs: `declare -n`

A nameref is a variable that acts as an *alias* for another variable — named at runtime. This is the clean, modern solution to passing arrays in and out of functions, and removes the need for the fragile `${!varname}` indirect expansion pattern in most cases.

> **Requires bash 4.3+.** Namerefs were introduced in bash 4.3 (2014). They are available on all modern Linux distributions. macOS system bash (3.2) does not support them.

Nameref basics

```
# declare -n creates a nameref — the variable IS the other variable
original="hello"
declare -n ref="original"

echo "$ref"                → hello
ref="world"               # modifying ref modifies original
echo "$original"           → world

# A nameref to an array — you get the full array
colours=( red green blue )
declare -n arr_ref="colours"
echo "${arr_ref[1]}"       → green
arr_ref+=( yellow )
echo "${colours[@]}"       → red green blue yellow
```

Namerefs in functions — the right way to return arrays

```
# Pass the name of the caller's variable; the function writes to it
get_stats() {
    local -n _result="$1"   # _result IS the caller's variable
    local total=0 count=0 min max
    shift                   # remaining args are the numbers

    min=$1; max=$1
    for n in "$@"; do
        (( total += n, count++ ))
        (( n < min )) && min=$n
        (( n > max )) && max=$n
    done

    # Write into an associative array via the nameref
    _result[count]=$count
    _result[total]=$total
```

```
    _result[min]=$min
    _result[max]=$max
    _result[mean]=$(echo "scale=2; $total/$count" | bc)
}

# Caller declares the output map first, then passes its name
declare -A stats
get_stats stats 5 10 3 8 15 2

printf "Count: %d  Total: %d  Min: %d  Max: %d  Mean: %s\n" \
    "${stats[count]}" "${stats[total]}" \
    "${stats[min]}"   "${stats[max]}"   "${stats[mean]}"
Count: 6  Total: 43  Min: 2  Max: 15  Mean: 7.16

# Returning an indexed array from a function
split_csv() {
    local -n _out="$1"
    IFS=',' read -r -a _out <<< "$2"
}
split_csv parts "alpha,beta,gamma,delta"
echo "${parts[2]}"          → gamma
```

Always name your nameref variables with a leading underscore (e.g. `_result`) inside functions. If the caller passes in a variable name that happens to match your local nameref name, bash will produce a circular reference error. The underscore prefix makes collisions essentially impossible in practice.

6 — Transformation Operators: ${var@operator}

Bash 4.4 added a family of transformation operators using the `@` suffix. These are less well-known but extremely useful for safe quoting, debugging, and code generation.

> **Requires bash 4.4+** (released 2016). Available on Ubuntu 18.04+, Debian 9+, and any modern distro. Run `echo $BASH_VERSION` to check.

OPERATOR	WHAT IT PRODUCES	EXAMPLE
`${var@Q}`	Quoted form — safe to re-use in shell input	`"hello world"` → `'hello world'`
`${var@E}`	Expands escape sequences like `$'...'`	`"tab:\t"` → `tab:`
`${var@P}`	Expanded as a prompt string (like PS1)	`"\u@\h"` → `alice@myhost`
`${var@A}`	Assignment form — declare statement to recreate the variable	`declare -- myvar='hello'`

OPERATOR	WHAT IT PRODUCES	EXAMPLE
`${var@a}`	Attribute flags of the variable	`r` for readonly, `A` for assoc array, etc.
`${var@U}`	Uppercase (alias for `${var^^}`)	*bash 5.1+*
`${var@u}`	First char uppercase (alias for `${var^}`)	*bash 5.1+*
`${var@L}`	Lowercase (alias for `${var,,}`)	*bash 5.1+*

@Q — safe quoting, the most practically useful operator

```
# @Q wraps the value in single quotes (or $'...' for special chars)
# making it safe to embed in eval, log output, or generated scripts

name="Alice O'Brien"
echo "${name@Q}"          → 'Alice O'\''Brien'   (properly escaped)

path="/home/alice/my documents"
echo "${path@Q}"          → '/home/alice/my documents'

# Log a command in a way that can be copy-pasted and re-run
log_command() {
    local quoted=()
    local arg
    for arg in "$@"; do
        quoted+=( "${arg@Q}" )
    done
    echo "[CMD] ${quoted[*]}" >&2
}
log_command cp "file with spaces.txt" "/dest/path"
[CMD] cp 'file with spaces.txt' '/dest/path'

# @A — get the declaration form of a variable
myvar="hello world"
echo "${myvar@A}"         → declare -- myvar='hello world'

declare -A config=( [host]=localhost [port]=5432 )
echo "${config@A}"        → declare -A config=([host]="localhost" [port]="5432" )
# This is exactly what declare -p produces — useful for serialising to a file
```

@a — inspecting variable attributes

```
declare -r  READONLY_VAR="fixed"
declare -i  INT_VAR=42
declare -a  INDEXED=( a b c )
declare -A  ASSOC=( [k]=v )
declare -x  EXPORTED="val"
declare -l  LOWER_VAR="INPUT"

echo "${READONLY_VAR@a}"   → r
echo "${INT_VAR@a}"        → i
echo "${INDEXED@a}"        → a
echo "${ASSOC@a}"          → A
echo "${EXPORTED@a}"       → x
echo "${LOWER_VAR@a}"      → l    (auto-lowercase attribute)

# Use @a in a guard to check a variable is the right type
if [[ "${myvar@a}" == *"A"* ]]; then
    echo "myvar is an associative array"
fi
```

7 — Compound and Nested Expansions

Parameter expansions can be nested and combined. The patterns below replace entire pipelines with single in-shell expressions — no subshell, no fork, no external process.

Chaining and nesting expansions

```
# ── Combining operations ──────────────────────────────

# Trim leading whitespace (no sed needed)
str="   hello world   "
trimmed="${str#"${str%%[! ]*}"}"    # strip leading spaces
echo "'$trimmed'"          → 'hello world   '

# Trim trailing whitespace
trimmed="${trimmed%"${trimmed##*[! ]}"}"
echo "'$trimmed'"          → 'hello world'

# ── Default with transformation ──────────────────────
# Uppercase the value, or use a default if unset
mode="production"
echo "${mode^^:-UNKNOWN}"  → PRODUCTION
```

```
# The above doesn't nest — you need two steps for that:
mode_upper="${mode:-unknown}"   # apply default first
echo "${mode_upper^^}"      → PRODUCTION

# ── Dynamic variable names via indirection ──────────────────
# Build variable names on the fly and look them up
region="eu"
eu_endpoint="https://eu.api.example.com"
us_endpoint="https://us.api.example.com"

endpoint_var="${region}_endpoint"
echo "${!endpoint_var}"    → https://eu.api.example.com

# ── Length of a transformed value ──────────────────────────
word="Hello"
echo "${#word}"            → 5
# You can't do ${#word^^} directly — assign to intermediate var
upper="${word^^}"
echo "${#upper}"           → 5  (same length, different case)

# ── Array expansion with per-element transformation ─────────
tags=( bash linux scripting )
# Uppercase every element
echo "${tags[@]^^}"        → BASH LINUX SCRIPTING
# Replace hyphens with underscores in every element
keys=( "my-key" "another-key" "third-key" )
echo "${keys[@]//-/_}"    → my_key another_key third_key
```

8 — Quick Reference

Default / assign / error / alternate

EXPANSION	EXPANDS TO	SIDE EFFECT
`${v:-val}`	*val* if v unset or empty, else v	none
`${v-val}`	*val* if v unset, else v	none
`${v:=val}`	*val* if v unset or empty, else v	assigns v=val
`${v=val}`	*val* if v unset, else v	assigns v=val
`${v:?msg}`	v (if set and non-empty)	error+exit if unset/empty

EXPANSION	EXPANDS TO	SIDE EFFECT
`${v?msg}`	v (if set)	error+exit if unset
`${v:+val}`	*val* if v set and non-empty, else empty	none
`${v+val}`	*val* if v set at all, else empty	none

String / substring operations

EXPANSION	RESULT
`${#v}`	Length of v in characters
`${v:n}`	Substring from position n to end
`${v:n:len}`	Substring: len chars from position n
`${v: -n}`	Last n characters (space before minus)
`${v#pat}`	Remove shortest prefix matching pat
`${v##pat}`	Remove longest prefix matching pat
`${v%pat}`	Remove shortest suffix matching pat
`${v%%pat}`	Remove longest suffix matching pat
`${v/pat/rep}`	Replace first match of pat with rep
`${v//pat/rep}`	Replace all matches of pat with rep
`${v/#pat/rep}`	Replace pat only at start
`${v/%pat/rep}`	Replace pat only at end
`${v^}`	Uppercase first char *bash 4.0+*
`${v^^}`	Uppercase all chars *bash 4.0+*
`${v,}`	Lowercase first char *bash 4.0+*
`${v,,}`	Lowercase all chars *bash 4.0+*

Indirection and transformation

EXPANSION	RESULT
`${!v}`	Value of the variable whose name is stored in v
`${!prefix@}`	Names of all variables starting with prefix

EXPANSION	RESULT
`${v@Q}`	Shell-quoted form of v *bash 4.4+*
`${v@A}`	declare statement to recreate v *bash 4.4+*
`${v@a}`	Attribute flags of v (r=readonly, i=integer, a=array…) *bash 4.4+*
`${v@E}`	Expand escape sequences in v *bash 4.4+*
`declare -n ref=name`	Make ref an alias for the variable named name *bash 4.3+*

🖍️ Exercises

Each exercise is designed to be solved purely with parameter expansion — resist the urge to reach for `sed`, `awk`, or `python`. The goal is to develop fluency with the shell's built-in string machinery.

EXERCISE 1

Write a function called `path_info()` that accepts a full file path as its only argument and — using only parameter expansion, no external commands — prints four lines: the directory, the filename (with extension), the base name (without any extension), and the extension (without the dot). Test it with `/var/log/nginx/access.log.2`.

Hint: directory = `${path%/*}`, filename = `${path##*/}`, extension = `${filename##*.}`, base = `${filename%%.*}`. Edge case: a file with no extension.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
set -euo pipefail

path_info() {
    local path="${1:?path_info: argument required}"

    local dir      filename    base    ext
    dir="${path%/*}"
    filename="${path##*/}"

    # Only split extension if there is a dot in the filename
    if [[ "$filename" == *"."* ]]; then
        ext="${filename##*.}"
```

```
        base="${filename%%.*}"
    else
        ext=""
        base="$filename"
    fi

    printf "Directory : %s\n" "$dir"
    printf "Filename  : %s\n" "$filename"
    printf "Base name : %s\n" "$base"
    printf "Extension : %s\n" "${ext:-(none)}"
}

path_info "/var/log/nginx/access.log.2"
Directory : /var/log/nginx
Filename  : access.log.2
Base name : access
Extension : log.2

path_info "/usr/bin/bash"
Directory : /usr/bin
Filename  : bash
Base name : bash
Extension : (none)
```

EXERCISE 2

Write a function called `return_multiple()` that calculates the quotient and remainder of two integers (passed as arguments), and returns *both* values to the caller using a nameref — so the caller can use them as separate variables. Also write a second function `swap()` that uses two namerefs to swap the values of two caller-side variables in place, without using a temporary variable visible to the caller.

Hint: for `return_multiple`, accept two nameref names as the first two arguments. For `swap`, use `local -n _a="$1" _b="$2"` and a local temp variable for the swap itself.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
set -euo pipefail
```

```bash
divmod() {
    # Usage: divmod QUOT_VAR REM_VAR DIVIDEND DIVISOR
    local -n _quot="$1"
    local -n _rem="$2"
    local _a="$3" _b="$4"
    [[ "$_b" -ne 0 ]] || { echo "divmod: division by zero" >&2; return 1; }
    _quot=$(( _a / _b ))
    _rem=$(( _a % _b ))
}

swap() {
    # Usage: swap VAR1 VAR2  — swaps values in place
    local -n _x="$1"
    local -n _y="$2"
    local _tmp="$_x"
    _x="$_y"
    _y="$_tmp"
}

# Test divmod
declare -i q r
divmod q r 17 5
echo "17 ÷ 5 = $q remainder $r"
17 ÷ 5 = 3 remainder 2

# Test swap
x="hello"; y="world"
echo "Before: x=$x  y=$y"
swap x y
echo "After:  x=$x  y=$y"
Before: x=hello  y=world
After:  x=world  y=hello
```

EXERCISE 3

Write a script called `config_check.sh` that uses `${var:?}` to validate a set of required environment variables (`APP_ENV`, `DB_HOST`, `DB_PORT`, `SECRET_KEY`). For optional variables (`LOG_LEVEL`, `TIMEOUT`), provide sensible defaults using `${var:-default}`. Then print a startup summary showing each variable's name, its value, and its source ("required", "default", or "env"). Use `${!prefix@}` to discover and print all `APP_` and `DB_` variables automatically.

Hint: store the "source" of each optional variable by checking before applying the default: `if [[ -n "${LOG_LEVEL+x}" ]]; then src="env"; else src="default"; fi`. Then apply the default. For the prefix listing, loop over `"${!APP_@}"` and `"${!DB_@}"`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# config_check.sh
set -uo pipefail   # no -e so :? error message is visible before exit

# ── Required variables — script exits here if any are unset/empty ──
APP_ENV="${APP_ENV:?APP_ENV is required (e.g. production, staging, dev)}"
DB_HOST="${DB_HOST:?DB_HOST is required}"
DB_PORT="${DB_PORT:?DB_PORT is required}"
SECRET_KEY="${SECRET_KEY:?SECRET_KEY is required}"

# ── Optional variables — track source before applying default ──
resolve_optional() {
    local -n _var="$1"
    local _default="$2"
    if [[ -n "${_var+x}" && -n "${_var}" ]]; then
        echo "env"
    else
        _var="$_default"
        echo "default"
    fi
}

ll_src=$(resolve_optional LOG_LEVEL "INFO")
to_src=$(resolve_optional TIMEOUT   "30")

# ── Print startup summary ────────────────────────────────────
echo
printf "\033[1mConfiguration Summary\033[0m\n"
printf '%.0s─' {1..50}; echo
printf "%-18s %-25s %s\n" "Variable" "Value" "Source"
printf '%.0s─' {1..50}; echo

# Required
for v in APP_ENV DB_HOST DB_PORT; do
    # Mask SECRET_KEY
```

```
    display="${!v}"
    printf "%-18s %-25s \033[32mrequired\033[0m\n" "$v" "$display"
done
# Show masked secret
masked="${SECRET_KEY:0:4}****${SECRET_KEY: -2}"
printf "%-18s %-25s \033[32mrequired\033[0m\n" "SECRET_KEY" "$masked"

# Optional
printf "%-18s %-25s \033[33m%s\033[0m\n" "LOG_LEVEL" "$LOG_LEVEL" "$ll_src"
printf "%-18s %-25s \033[33m%s\033[0m\n" "TIMEOUT"   "$TIMEOUT"   "$to_src"

printf '%.0s─' {1..50}; echo
echo
echo "All APP_ and DB_ variables in scope:"
for v in "${!APP_@}" "${!DB_@}"; do
    printf "  %s=%s\n" "$v" "${!v}"
done
```

EXERCISE 4

Write a function called `safe_log_args()` that takes any number of arguments (as a command and its arguments would be passed to it), and logs them in a copy-pasteable form using `${arg@Q}` to handle values containing spaces, quotes, or special characters. Then write a wrapper function `run()` that calls `safe_log_args` before executing its arguments as a command. Demonstrate it wrapping `cp` and `mkdir` with tricky paths.

Hint: in `safe_log_args()`, loop over `"$@"` and collect `"${arg@Q}"` into an array, then print the array joined with spaces. In `run()`, call `safe_log_args "$@"` first, then `"$@"` to execute. Test with paths containing spaces.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
set -euo pipefail

safe_log_args() {
    local quoted=()
    local arg
    for arg in "$@"; do
        quoted+=( "${arg@Q}" )
    done
```

```
    # Print as a reproducible shell command
    printf '\033[36m[RUN]\033[0m %s\n' "${quoted[*]}" >&2
}

run() {
    safe_log_args "$@"
    "$@"
}

# Demonstration
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

# These commands have spaces and special chars in paths
run mkdir -p "${TMPDIR}/my project/src files"
run touch "${TMPDIR}/my project/src files/main script.sh"
run cp "${TMPDIR}/my project/src files/main script.sh" \
       "${TMPDIR}/backup of main script.sh"

[RUN] mkdir -p '/tmp/tmp.abc123/my project/src files'
[RUN] touch '/tmp/tmp.abc123/my project/src files/main script.sh'
[RUN] cp '/tmp/tmp.abc123/my project/src files/main script.sh' '/tmp/tmp.abc123/bac

# The log output is safe to copy-paste and re-run exactly
```

🔀 Intermediate Topic 2 — Process Substitution and Advanced Redirection

The beginner course covered `>`, `>>`, `<`, `2>&1`, and pipelines. That's enough to get things done, but the moment you need to compare two command outputs, write to a log file and the terminal at the same time, or redirect inside a long-lived loop without closing and reopening files, you hit the limits of the basics. This chapter covers process substitution, named pipes, `exec`-based persistent file descriptors, and every redirection form bash supports — the machinery that serious scripts are built on.

1 — How Redirection Works Under the Hood

Every process inherits three open file descriptors: **stdin (0)**, **stdout (1)**, and **stderr (2)**. A redirection operator doesn't move data — it *rewires* file descriptor numbers before the command starts, using the kernel's `dup2()` system call. Understanding this makes every redirection form predictable.

```
Without redirection:
  script ──FD 0──▶ /dev/pts/0   (keyboard)
         ──FD 1──▶ /dev/pts/0   (terminal)
         ──FD 2──▶ /dev/pts/0   (terminal)

After:  cmd > file.txt 2>&1
  Step 1: open file.txt → FD 1   (stdout now points to file.txt)
  Step 2: dup FD 1 to FD 2       (stderr now also points to file.txt)

  cmd    ──FD 0──▶ /dev/pts/0   (keyboard, unchanged)
         ──FD 1──▶ file.txt
         ──FD 2──▶ file.txt     (FD 2 is a copy of FD 1's target)

Order matters!  cmd 2>&1 > file.txt  is WRONG — copies stderr→terminal THEN redirects stdou
```

The classic ordering mistake: `cmd 2>&1 >file.txt` does *not* send both streams to the file. Redirections are processed left-to-right. At the moment `2>&1` is evaluated, FD `1` still points to the terminal, so stderr ends up on the terminal and only stdout goes to the file. Always write `cmd >file.txt 2>&1` (redirect stdout first, then copy that to stderr).

Every redirection form at a glance

SYNTAX	WHAT IT DOES
`cmd > file`	Redirect stdout to file (truncate)
`cmd >> file`	Redirect stdout to file (append)
`cmd < file`	Read stdin from file
`cmd 2> file`	Redirect stderr to file
`cmd &> file`	Redirect both stdout and stderr to file (bash shorthand)
`cmd &>> file`	Append both stdout and stderr to file
`cmd > file 2>&1`	Redirect stdout to file, then copy that to stderr (POSIX-safe form)
`cmd 2>&1 > file`	Copy stderr to terminal, then redirect stdout to file (usually a bug)
`cmd > /dev/null`	Discard stdout
`cmd 2> /dev/null`	Discard stderr
`cmd &> /dev/null`	Discard all output
`cmd >&2`	Send stdout to wherever stderr currently points
`cmd << EOF ... EOF`	Here-document: feed multi-line string as stdin
`cmd <<< "string"`	Here-string: feed single string as stdin
`cmd n> file`	Open file on file descriptor n for writing
`cmd n< file`	Open file on file descriptor n for reading
`cmd n>&m`	Duplicate FD m to FD n
`cmd n>&-`	Close file descriptor n
`cmd {var}> file`	Open file on a bash-chosen FD; store number in var (bash 4.1+)

2 — Here-Documents and Here-Strings

Here-documents are the clean way to feed multi-line text to a command without a temporary file. There are four variations, each with different quoting and indentation behaviour.

The four here-doc forms

```
# ── 1. Standard: variable expansion IS performed ──────────────
name="Alice"
```

```
cat << EOF
Hello, $name!
Today is $(date +%A).
EOF
Hello, Alice!
Today is Monday.

# ── 2. Quoted delimiter: NO expansion (literal text) ──────────
cat << 'EOF'
Hello, $name!    <- this prints literally, no substitution
EOF
Hello, $name!

# ── 3. Indented form (<<-): strips leading TABS only ──────────
# Use tabs (not spaces) for the heredoc lines
if true; then
        cat <<- EOF
        This line is indented with a tab — the tab is stripped.
        So is this one.
        EOF
fi
This line is indented with a tab — the tab is stripped.
So is this one.

# ── 4. Here-string: single value, no newlines ─────────────────
# Avoids echo | cmd  which spawns a subshell
read -r first rest <<< "one two three"
echo "first=$first  rest=$rest"
first=one  rest=two three

# bc without a pipe
result=$(bc <<< "scale=4; 355/113")
echo "$result"              → 3.1415

# Passing a variable as stdin to a command expecting a file
json='{"key":"value"}'
jq '.key' <<< "$json"     → "value"
```

Here-strings (<<<) always append a newline to the string before feeding it to the command — most commands that read stdin expect this, but be aware if you're doing byte-level work.

Here-docs for generating files and scripts

Writing multi-line content to a file

```
# Write an entire config file in one heredoc
db_host="db.example.com"
db_port=5432

cat > /etc/myapp/config.ini << EOF
[database]
host = ${db_host}
port = ${db_port}
name = myapp

[server]
port = 8080
workers = 4
EOF

# Generate a script with literal dollar signs (quoted delimiter)
cat > /tmp/generated.sh << 'SCRIPT'
#!/usr/bin/env bash
echo "Today is $(date)"
echo "User: $USER"
SCRIPT
chmod +x /tmp/generated.sh
# The $USER and $(date) will expand when generated.sh runs, not now

# Append to a file using tee (allows both write and stdout if needed)
tee -a /var/log/myapp.log << EOF
[$(date -Is)] Application started
[$(date -Is)] PID: $$
EOF
```

3 — Process Substitution: `<(cmd)` and `>(cmd)`

Process substitution is one of bash's most powerful features and one of the least understood. It lets you use a command's output *as if it were a file*, without creating a temporary file. Bash runs the command in the background, connects it to a named file descriptor under `/dev/fd/`, and passes that path to the outer command.

```
How  diff <(sort a.txt) <(sort b.txt)  works:

  bash creates two processes:
    [sort a.txt] ──writes──▶ /dev/fd/63  (a kernel pipe)
```

```
  [sort b.txt] ──writes──▶ /dev/fd/62

diff is called as:
  diff /dev/fd/63 /dev/fd/62

diff reads from those paths like files.
No temp files created.  All three processes run concurrently.
```

> **Process substitution is bash-specific.** It requires `#!/usr/bin/env bash` — it is not available in POSIX `sh`, `dash`, or BusyBox shells. If portability matters, use named pipes (covered in section 5) instead.

Input process substitution — <(cmd)

Using command output as a file argument

```bash
# ── Compare two sorted files without modifying them ──────────
diff <(sort old.txt) <(sort new.txt)

# ── Find lines present in one file but not another ───────────
comm -23 <(sort list_a.txt) <(sort list_b.txt)
# comm requires sorted input — process substitution sorts on-the-fly

# ── Join two CSVs on a common field ──────────────────────────
join -t, -1 1 -2 1 \
    <(sort -t, -k1,1 users.csv) \
    <(sort -t, -k1,1 orders.csv)

# ── Read from a process substitution with while ──────────────
# This is the key fix for the "pipe loop loses variables" problem
# (covered in depth in Topic 4 — Subshells)
count=0
while IFS= read -r line; do
    (( count++ ))
done < <(find /var/log -name '*.log')
echo "Found $count log files"
# Compare the broken version:  find ... | while ... done  ← $count resets to 0

# ── Read output of two commands with a single while loop ─────
while IFS= read -r -u3 a && IFS= read -r -u4 b; do
    echo "$a  |  $b"
done 3< <(cat file1.txt) \
     4< <(cat file2.txt)
```

The space in `< <(cmd)` is required — the first `<` is the stdin redirect; the second is the start of the process substitution. Without the space, bash sees `<<` which starts a here-document.

Output process substitution — `>(cmd)`

Less commonly used, output process substitution feeds a command's stdin from another command's stdout — allowing a command that only writes to stdout to be "directed" into multiple consumers simultaneously.

Writing to multiple destinations with tee + >(cmd)

```
# ── Write to a file AND pass through to another command ──────
# tee sends its stdin to both the file and the process substitution
generate_report | tee >(gzip > report.gz) | mail -s "Report" boss@example.com
# One pipeline: compress to file, AND email — data read only once

# ── Split stderr and stdout to different log files ──────────
cmd="./my_script.sh"
$cmd \
    1> >(tee -a stdout.log) \
    2> >(tee -a stderr.log >&2)
# stdout goes to stdout.log AND terminal; stderr to stderr.log AND terminal

# ── Log with timestamps in real time ────────────────────────
LOGFILE="app.log"
timestamp_logger() {
    while IFS= read -r line; do
        printf '[%s] %s\n' "$(date '+%H:%M:%S')" "$line"
    done
}

./my_app > >(timestamp_logger > "$LOGFILE")

# ── Fan-out: send output to three places simultaneously ──────
./my_app | tee \
    >(grep ERROR > errors.log) \
    >(gzip > full_output.gz) \
    > stdout.log
```

4 — exec and Persistent File Descriptors

When you write `cmd > file`, the file is opened for the duration of that one command. If you're inside a loop that writes thousands of times, each iteration opens and closes the file. Using `exec` to redirect a file descriptor in the *current shell* keeps the file open for the entire script — dramatically faster for heavy I/O, and essential for any pattern that needs to interleave reading and writing.

exec redirects — persistent FDs for the current shell

```
# ── Redirect all stdout for the rest of the script ──────────
exec > output.log      # everything printed after this goes to output.log
echo "This goes to the file"
ls -la                 # stdout of ls also goes to the file

# ── Redirect stderr to a log file for the rest of the script ──
exec 2> errors.log
echo "normal output"   # still to terminal (stdout unchanged)
ls /nonexistent        # error message goes to errors.log

# ── Save and restore FDs — the classic pattern ──────────────
# Save current stdout to FD 3, redirect stdout to file, then restore
exec 3>&1              # save stdout (1) into FD 3
exec > capture.log     # redirect stdout to file
echo "in the file"
exec 1>&3              # restore stdout from FD 3
exec 3>&-              # close FD 3 (no longer needed)
echo "back to terminal"

# ── Open a file for reading on a custom FD ──────────────────
exec 4< data.txt        # open data.txt for reading on FD 4
while IFS= read -r -u4 line; do
    # read -u4 reads from FD 4 instead of stdin
    # stdin is still free for interactive prompts inside the loop
    read -r -p "Process '$line'? [y/n] " ans < /dev/tty
    [[ "$ans" == "y" ]] && echo "Processing: $line"
done
exec 4<&-              # close FD 4 when done
```

Auto-assigned file descriptors — {var}> file

Let bash choose the FD number (bash 4.1+)

```
# Hard-coding FD numbers risks collision with FDs opened by bash itself
# {var} lets bash pick a free number and stores it in var
```

```
exec {logfd}> app.log       # bash chooses FD (typically 10+), stores in $logfd
echo "Log FD is: $logfd"  → Log FD is: 10

# Write to the log using the variable
echo "Application starting" >&$logfd
echo "Another log line"     >&$logfd

# Close when done
exec {logfd}>&-

# Practical: a logging function using a persistent FD
LOGFILE="run_$(date +%Y%m%d_%H%M%S).log"
exec {LOG}>> "$LOGFILE"    # append mode

log() {
    printf '[%s] %s\n' "$(date '+%T')" "$*" >&$LOG
}

log "Script started"
log "Running as: $USER"
# ... script body ...
exec {LOG}>&-              # close log at end
```

Using {var} is safer than picking a fixed number like 3 or 4 because bash uses higher-numbered FDs internally and won't assign one that's already in use.

Swapping stdout and stderr

The three-step FD swap

```
# Swap stdout and stderr (send stdout to stderr and vice-versa)
# Uses the same save/swap/restore pattern as a variable swap
{ some_command; } 3>&1 1>&2 2>&3
# Step 1: save stdout to FD 3        (3>&1)
# Step 2: point stdout at stderr      (1>&2)
# Step 3: point stderr at saved FD 3  (2>&3)

# Practical: capture stderr into a variable while stdout passes through
stderr_output=$( { some_command; } 2>&1 1>&3 ) 3>&1
# $() captures FD 1; the swap routes stderr there and stdout to the terminal
echo "Captured stderr: $stderr_output"

# Capture stdout AND stderr into separate variables simultaneously
exec {stdout_fd}>/tmp/stdout.$$
exec {stderr_fd}>/tmp/stderr.$$
```

```
some_command >&$stdout_fd 2>&$stderr_fd
rc=$?
exec {stdout_fd}>&-
exec {stderr_fd}>&-
out=$(cat /tmp/stdout.$$)
err=$(cat /tmp/stderr.$$)
rm -f /tmp/stdout.$$ /tmp/stderr.$$
echo "Exit: $rc | Out: $out | Err: $err"
```

5 — Named Pipes (FIFOs)

A named pipe (FIFO — First In, First Out) is like a regular pipe but with a name in the filesystem. Unlike a | pipeline, a FIFO can connect processes that aren't related, or that start at different times. A write to a FIFO blocks until a reader opens it, and vice versa — this built-in synchronisation makes FIFOs useful for coordinating producer/consumer workflows.

mkfifo — creating and using named pipes

```
# Create a named pipe
mkfifo /tmp/mypipe
ls -l /tmp/mypipe
prw-r--r-- 1 alice alice 0 Jun  9 09:00 /tmp/mypipe
# The 'p' in 'prw-' means "pipe"

# Terminal 1 — writer (blocks until reader connects)
echo "hello from the other side" > /tmp/mypipe

# Terminal 2 — reader
cat /tmp/mypipe
hello from the other side

# Remove when done — FIFOs persist until deleted
rm /tmp/mypipe
```

FIFO as a portable alternative to process substitution

```
# Process substitution equivalent using a FIFO (POSIX-portable)
fifo=$(mktemp -u)        # generate a unique path without creating the file
mkfifo "$fifo"
```

```
trap 'rm -f "$fifo"' EXIT

# Start the producer in the background, writing to the FIFO
sort input.txt > "$fifo" &
# Consumer reads from the FIFO
uniq "$fifo"
# The FIFO synchronises them — uniq waits for sort to write

# ── Multi-producer / single-consumer ──────────────────────
# Several background jobs write to the same FIFO; one reader collects
results=$(mktemp -u)
mkfifo "$results"
trap 'rm -f "$results"' EXIT

for host in host1 host2 host3; do
    ( ssh "$host" 'uptime'; echo "$host done" ) > "$results" &
done

# Collect all results — read until all writers are done
cat "$results"
wait
```

A key difference from anonymous pipes: FIFOs have names so they can be opened multiple times by independent processes. Each open/close pair is independent, but data flows one way and in order, just like any pipe.

Using a FIFO as a persistent input channel (log sink)

A log aggregator — multiple writers, one reader

```
#!/usr/bin/env bash
# log_server.sh — reads from a FIFO and writes timestamped lines to a log
set -euo pipefail

FIFO=/tmp/app_log.fifo
LOGFILE=/var/log/app_combined.log

mkfifo -m 600 "$FIFO"
trap 'rm -f "$FIFO"' EXIT INT TERM

# Open the FIFO for reading — keep it open even when no writer is connected
# Opening with O_RDWR (/dev/stdin trick) prevents EOF when last writer leaves
exec {fd}<> "$FIFO"         # <> opens for both read and write

echo "Log server started. FIFO: $FIFO"
```

```
while IFS= read -r -u"$fd" line; do
    printf '[%s] %s\n' "$(date -Iseconds)" "$line" >> "$LOGFILE"
done

# Writers send to the FIFO like this:
# echo "worker A: job complete" > /tmp/app_log.fifo
```

6 — Special Paths: /dev/stdin, /dev/fd/N, /dev/tcp

Shell-special paths for redirection

```
# /dev/stdin, /dev/stdout, /dev/stderr
# Symbolic names for FDs 0, 1, 2 — useful when a command expects a filename
wc -l /dev/stdin <<< "line 1\nline 2\nline 3"
3 /dev/stdin

# Useful when a command requires a filename but you have a string
md5sum /dev/stdin <<< "hash this"

# /dev/null — the bitbucket
# Redirect stderr to /dev/null to suppress error messages
ls /nonexistent 2> /dev/null
# Run a command silently (discard all output)
some_noisy_cmd &> /dev/null

# /dev/fd/N — access any open file descriptor by number
# Used automatically by process substitution: <(cmd) → /dev/fd/63
ls -la /dev/fd/            # see which FDs are open in the current shell

# /dev/tcp/HOST/PORT — raw TCP connection (bash built-in, no netcat needed)
# Opens a TCP socket; bash handles it as a regular file descriptor
exec 3<> /dev/tcp/example.com/80
# Send an HTTP request
printf 'GET / HTTP/1.0\r\nHost: example.com\r\n\r\n' >&3
# Read the response
cat <&3
exec 3<&-

# Simple port-open check using /dev/tcp
port_open() {
    local host="$1" port="$2"
    (exec 3<> /dev/tcp/"${host}/${port}") 2>/dev/null
```

```
}
if port_open "db.example.com" 5432; then
    echo "Database port is open"
else
    echo "Database unreachable"
fi
```

/dev/tcp is a bash feature, not a real filesystem path. It only works in bash (not sh, dash, or zsh). Some security-hardened systems disable it at compile time.

7 — tee and Multi-Destination Output

`tee` reads from stdin and writes to both stdout *and* one or more files simultaneously. Combined with process substitution and output FDs, it's the glue that makes fan-out pipelines possible.

tee patterns — display AND capture, branching pipelines

```
# Basic: show output on terminal and save to a file
./long_running_job | tee job.log

# Append to existing file
./job | tee -a job.log

# ── Capture output into a variable while still showing it ──────
output=$(./job | tee /dev/tty)
# /dev/tty is the controlling terminal — writes bypass stdout redirect

# ── Fan out to multiple files ──────────────────────────────────
./job | tee copy1.log copy2.log copy3.log > /dev/null
# Writes to all three files; /dev/null discards the tee stdout

# ── Pipeline inspection — "T" into the middle of a pipe ────────
# See what's flowing through a pipeline at any point
cat input.txt \
    | tee /tmp/after_cat.log \
    | grep 'ERROR' \
    | tee /tmp/after_grep.log \
    | sort -u

# ── Use tee to copy a download to multiple destinations ────────
curl -sL https://example.com/bigfile.tar.gz \
    | tee /backups/bigfile.tar.gz \
    | tar -xz -C /opt/app/
```

```
# Saves the archive AND extracts it in one download pass

# ── Fan-out with different processing per branch ──────────────
./my_app | tee \
    >(grep -i 'error\|warn' >> alerts.log) \
    >(gzip -9 > full.log.gz) \
    >(wc -l > line_count.txt) \
    > /dev/null
```

8 — Quick Reference

Process substitution forms

SYNTAX	DESCRIPTION	WHERE CMD RUNS
`<(cmd)`	Provide cmd's stdout as a readable file path	Background subshell
`>(cmd)`	Provide a writable file path that feeds cmd's stdin	Background subshell
`cmd1 | tee >(cmd2)`	Send cmd1 output to both cmd2 and stdout	cmd2 in background

Key exec redirection patterns

PATTERN	EFFECT
`exec > file`	Redirect all script stdout to file from this point
`exec 2> file`	Redirect all script stderr to file from this point
`exec 3>&1`	Save FD 1 (stdout) into FD 3
`exec 1>&3`	Restore FD 1 from FD 3
`exec 3>&-`	Close FD 3
`exec 4< file`	Open file for reading on FD 4
`exec {fd}> file`	Open file on auto-chosen FD; number stored in $fd
`exec 3<> file`	Open file for both reading and writing on FD 3

FIFO quick reference

COMMAND	DESCRIPTION
`mkfifo /path/to/fifo`	Create a named pipe
`mkfifo -m 600 /path/to/fifo`	Create with restricted permissions
`mktemp -u`	Generate a unique path without creating a file (use with mkfifo)
`exec {fd}<> fifo`	Keep FIFO open (prevents EOF when last writer disconnects)

🖍️ Exercises

These exercises require combining multiple techniques from this chapter. Resist reaching for temporary files unless explicitly building one — the goal is fluency with pipelines, process substitution, and persistent file descriptors.

EXERCISE 1

Write a script called `compare_dirs.sh` that accepts two directory paths as arguments and reports three things: (1) files present in the first directory but not the second, (2) files present in the second but not the first, and (3) files present in both. Use `comm` with process substitution — no temporary files. Handle the case where either path doesn't exist.

Hint: `comm` requires sorted input. `find DIR -maxdepth 1 -type f -printf '%f\n' | sort` gives sorted filenames. Use `comm -23` for only-in-first, `comm -13` for only-in-second, `comm -12` for in-both.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# compare_dirs.sh  DIR1  DIR2
set -euo pipefail

dir1="${1:?Usage: compare_dirs.sh DIR1 DIR2}"
dir2="${2:?Usage: compare_dirs.sh DIR1 DIR2}"

[[ -d "$dir1" ]] || { echo "Error: '$dir1' is not a directory" >&2; exit 1; }
[[ -d "$dir2" ]] || { echo "Error: '$dir2' is not a directory" >&2; exit 1; }

# Helper: sorted filenames in a directory (one per line)
sorted_files() {
    find "$1" -maxdepth 1 -type f -printf '%f\n' | sort
}
```

```
printf '\033[1mComparing:\033[0m\n  %s\n  %s\n\n' "$dir1" "$dir2"

# Only in dir1
printf '\033[33mOnly in %s:\033[0m\n' "$dir1"
comm -23 <(sorted_files "$dir1") <(sorted_files "$dir2") \
    | sed 's/^/  /'

# Only in dir2
printf '\033[33mOnly in %s:\033[0m\n' "$dir2"
comm -13 <(sorted_files "$dir1") <(sorted_files "$dir2") \
    | sed 's/^/  /'

# In both
printf '\033[32mIn both:\033[0m\n'
comm -12 <(sorted_files "$dir1") <(sorted_files "$dir2") \
    | sed 's/^/  /'
```

EXERCISE 2

Write a function called `run_logged()` that executes any command passed to it and: (a) shows stdout in the terminal in real time, (b) simultaneously writes stdout to a timestamped log file, (c) captures stderr separately and prints a summary at the end if there were any errors, and (d) returns the original exit code of the command. Do not use temporary files for the main stdout flow — use process substitution and `tee`.

Hint: use `tee >(timestamp_fn >> logfile)` to simultaneously print and log. For stderr, redirect to a temp file (`mktemp`) so you can display it at the end. Use a subshell group to capture the exit code: `{ cmd; } ... ; rc=$?`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
set -uo pipefail

run_logged() {
    local logfile="run_$(date +%Y%m%d_%H%M%S)_$$.log"
    local errfile
    errfile=$(mktemp)

    local rc=0
```

```
    # Timestamper: prepend HH:MM:SS to each line
    _stamp() {
        while IFS= read -r line; do
            printf '[%s] %s\n' "$(date '+%T')" "$line"
        done
    }

    printf '\033[36m[RUN]\033[0m %s\n' "$*"
    printf '\033[2m[LOG] → %s\033[0m\n' "$logfile"

    # Run command: stdout fans out to terminal + timestamped logfile
    #              stderr goes to errfile
    "$@" \
        2> "$errfile" \
        | tee >(_stamp >> "$logfile") \
        || rc=$?

    # If there was stderr output, show it
    if [[ -s "$errfile" ]]; then
        printf '\n\033[31m[STDERR]\033[0m\n' >&2
        cat "$errfile" >&2
        # Also append stderr to log
        _stamp < "$errfile" >> "$logfile"
    fi
    rm -f "$errfile"

    if (( rc == 0 )); then
        printf '\033[32m[OK]\033[0m Exit 0\n'
    else
        printf '\033[31m[FAIL]\033[0m Exit %d\n' "$rc" >&2
    fi
    return $rc
}

# Test
run_logged ls -la /etc/hosts /nonexistent
```

EXERCISE 3

Write a script called `multi_log.sh` that uses `exec` with auto-assigned file descriptors (`{var}> file`) to open three log files at the start — `info.log`, `warn.log`, `error.log` — and a function `log LEVEL MESSAGE` that writes to the appropriate file descriptor based on the level. The script should process a list of simulated events, close all file descriptors when done, and print a summary of how many lines went to each log. Do not reopen the files inside the loop.

Hint: store the FD numbers in variables `INFO_FD`, `WARN_FD`, `ERROR_FD` using `exec {INFO_FD}> info.log`. In the log function, use `printf ... >&$INFO_FD` etc. Use a `case` statement to dispatch on level. For the count, use `wc -l` on each file after closing the FDs.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# multi_log.sh
set -euo pipefail

LOGDIR="$(mktemp -d)"
trap 'echo "Logs in: $LOGDIR"' EXIT

# Open three log files on auto-assigned FDs
exec {INFO_FD}>  "${LOGDIR}/info.log"
exec {WARN_FD}>  "${LOGDIR}/warn.log"
exec {ERROR_FD}> "${LOGDIR}/error.log"

log() {
    local level="${1^^}"    # uppercase the level
    local msg="$2"
    local ts
    ts="$(date '+%H:%M:%S')"

    case "$level" in
        INFO)  printf '[%s] INFO  %s\n'  "$ts" "$msg" >&$INFO_FD  ;;
        WARN)  printf '[%s] WARN  %s\n'  "$ts" "$msg" >&$WARN_FD  ;;
        ERROR) printf '[%s] ERROR %s\n'  "$ts" "$msg" >&$ERROR_FD ;;
        *)     printf '[%s] ??    %s\n'  "$ts" "$msg" >&$INFO_FD  ;;
    esac
}

# Simulated events — in a real script this would be a processing loop
log INFO  "Application starting"
log INFO  "Loading configuration"
```

```
log WARN  "Config file not found, using defaults"
log INFO  "Connected to database"
log INFO  "Processing 1000 records"
log WARN  "Record 42 has missing field 'email'"
log ERROR "Failed to insert record 99: duplicate key"
log INFO  "Batch complete"
log ERROR "Connection reset during final commit"

# Close all file descriptors
exec {INFO_FD}>&-
exec {WARN_FD}>&-
exec {ERROR_FD}>&-

# Summary
echo
printf "%-12s %s\n" "Log" "Lines"
printf '%.0s─' {1..20}; echo
for f in info warn error; do
    n=$(wc -l < "${LOGDIR}/${f}.log")
    printf "%-12s %d\n" "${f}.log" "$n"
done
```

EXERCISE 4

Write a script called `health_check.sh` that checks whether a set of services are reachable using `/dev/tcp`. It should accept a list of `host:port` pairs (either as arguments or from a here-string), check each one with a configurable timeout, and produce a report showing each service as UP or DOWN with response time in milliseconds. Use a FIFO or process substitution to feed the list if it comes from a here-doc.

Hint: use `( exec 3<> /dev/tcp/"$host"/"$port" ) 2>/dev/null` in a subshell so the socket closes on exit. Capture `$SECONDS` or `date +%s%N` before and after for timing. For timeout, wrap the subshell in a background job and `wait` with a timeout loop or use `timeout 2 bash -c "..."`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# health_check.sh  [host:port ...]
set -uo pipefail
```

```
TIMEOUT=${TIMEOUT:-3}

check_service() {
    local host="$1" port="$2"
    local start_ns end_ns ms

    start_ns=$(date '+%s%N')

    if timeout "$TIMEOUT" bash -c \
        "exec 3<> /dev/tcp/${host}/${port}" 2>/dev/null; then
        end_ns=$(date '+%s%N')
        ms=$(( (end_ns - start_ns) / 1000000 ))
        printf '  \033[32m%-6s\033[0m %-30s %dms\n' "UP" "${host}:${port}" "$ms"
    else
        printf '  \033[31m%-6s\033[0m %-30s (timeout ${TIMEOUT}s)\n' "DOWN" "${host
    fi
}

printf '\033[1mService Health Check\033[0m  (timeout: %ss)\n' "$TIMEOUT"
printf '%.0s─' {1..50}; echo

if (( $# > 0 )); then
    # Args mode: each arg is host:port
    for svc in "$@"; do
        host="${svc%:*}"
        port="${svc##*:}"
        check_service "$host" "$port"
    done
else
    # Stdin mode: read host:port from process substitution / pipe
    while IFS=: read -r host port; do
        [[ -z "$host" || "$host" == '#'* ]] && continue
        check_service "$host" "$port"
    done
fi



# Usage examples:
# ./health_check.sh google.com:443 github.com:22 localhost:5432
#
# Or feed a list via here-string:
# ./health_check.sh <<< $'google.com:80\ngoogle.com:443\nlocalhost:5432'
```

```
#
# Or via process substitution:
# ./health_check.sh < <(grep -v '^#' services.txt)
```

📡 Intermediate Topic 3 — Job Control and Signals

Every process in Linux lives inside a web of signals — the operating system's notification mechanism for events ranging from "the user pressed Ctrl+C" to "your parent process died" to "please flush your buffers and exit gracefully." Most scripts ignore this entirely and work fine for simple tasks. But the moment your script spawns background jobs, holds open files, manages a lock, or does anything that matters, you need to understand signals and job control: how to catch them, forward them to child processes, clean up properly, and implement graceful shutdown. This chapter covers the full picture.

1 — What Are Signals?

A signal is an asynchronous notification sent to a process by the kernel, another process, or the process itself. When a signal arrives, the process's current execution is interrupted and a signal handler runs — either a default kernel action or a custom function you register with `trap`. Signals have numbers, but are almost always referred to by name.

```
Signal delivery model:

  Sender                    Kernel                    Target process
  ──────                    ──────                    ──────────────
  kill -TERM 1234    ──────▶ marks signal pending ──────▶  interrupted
  Ctrl+C in terminal ──────▶ SIGINT to process group       handler runs
  process exit       ──────▶ SIGHUP to children            (or default action)

Default actions for unhandled signals:
  Terminate   — kill the process                    (SIGTERM, SIGHUP, SIGPIPE…)
  Terminate+core — kill and write a core dump       (SIGSEGV, SIGABRT…)
  Stop        — pause the process (resumable)        (SIGTSTP, SIGSTOP)
  Continue    — resume a stopped process             (SIGCONT)
  Ignore      — silently discard                      (SIGCHLD by default in some shells)
```

Signal reference table

SIGNAL	NUMBER	DEFAULT ACTION	COMMON CAUSE / MEANING
SIGHUP	1	Terminate	Terminal closed; also used to ask daemons to reload config
SIGINT	2	Terminate	Ctrl+C — interactive interrupt
SIGQUIT	3	Core dump	Ctrl+\ — quit and dump core

SIGNAL	NUMBER	DEFAULT ACTION	COMMON CAUSE / MEANING
SIGKILL	9	Terminate	Unconditional kill — cannot be caught, blocked, or ignored
SIGUSR1	10	Terminate	Application-defined; use it for custom notifications
SIGUSR2	12	Terminate	Application-defined; second user signal
SIGPIPE	13	Terminate	Write to a pipe with no reader (broken pipe)
SIGALRM	14	Terminate	Timer set with `alarm()` expired
SIGTERM	15	Terminate	Polite termination request — the default signal sent by `kill`
SIGCHLD	17	Ignore	Child process stopped or terminated
SIGCONT	18	Continue	Resume a stopped process
SIGSTOP	19	Stop	Pause — cannot be caught or ignored
SIGTSTP	20	Stop	Ctrl+Z — terminal stop (can be caught, unlike SIGSTOP)
SIGWINCH	28	Ignore	Terminal window resized

Listing signals on your system

```
kill -l                  # list all signal names
kill -l TERM             # get number for SIGTERM  → 15
kill -l 9                # get name for signal 9   → KILL
trap -l                  # bash's trap list (includes pseudo-signals)
```

2 — `trap` In Depth

`trap` registers a handler — a shell command or function — to run when a specific signal or event is received. It is the primary mechanism for cleanup, graceful shutdown, and debugging in bash scripts.

trap syntax and basic usage

```
# Syntax:  trap 'COMMAND' SIGNAL [SIGNAL ...]
# The command is a string — evaluated when the signal fires, not when trap runs

# ── Catch Ctrl+C ──────────────────────────────────────
trap 'echo "Caught Ctrl+C — cleaning up"; exit 1' INT
```

```
# ── Run cleanup on any exit — the most important trap ──────────
trap 'rm -f /tmp/mylock.$$' EXIT
# EXIT fires on: normal exit, exit N, error with set -e, signals
# It does NOT fire on SIGKILL (nothing can catch that)

# ── Reset a trap to its default behaviour ──────────────────────
trap - INT            # reset SIGINT to default (terminate)

# ── Ignore a signal ────────────────────────────────────────────
trap '' HUP           # ignore SIGHUP (e.g. when run via nohup)
trap '' PIPE          # ignore broken pipe — useful in producers

# ── View currently registered traps ────────────────────────────
trap -p               # print all active traps
trap -p EXIT          # print just the EXIT trap
```

Bash pseudo-signal events

In addition to real signals, bash supports three special event names usable with `trap`:

EVENT	WHEN IT FIRES
`EXIT`	When the shell exits for any reason (not SIGKILL). Runs after the last command.
`ERR`	When any command exits with a non-zero status (and `set -e` is not set, or is set and would trigger). Useful for centralised error logging.
`DEBUG`	Before every simple command. Used for tracing and step-by-step debuggers.
`RETURN`	When a function or sourced file returns.

The canonical EXIT trap pattern

Clean, robust cleanup with EXIT trap

```
#!/usr/bin/env bash
set -euo pipefail

# Declare resources upfront so cleanup is always safe to call
TMPDIR=""
LOCKFILE=""
CHILD_PID=""

cleanup() {
    local rc=$?        # capture exit code before any cleanup commands change it
```

```
    # Remove temp dir if it was created
    [[ -n "$TMPDIR"   && -d "$TMPDIR"   ]] && rm -rf "$TMPDIR"
    # Release lock if held
    [[ -n "$LOCKFILE" && -f "$LOCKFILE" ]] && rm -f  "$LOCKFILE"
    # Kill background child if still running
    [[ -n "$CHILD_PID" ]] && kill "$CHILD_PID" 2>/dev/null || true
    exit "$rc"          # preserve the original exit code
}
trap cleanup EXIT

# ── Script body ──────────────────────────────────────────
TMPDIR=$(mktemp -d)
LOCKFILE=/var/run/myscript.lock
touch "$LOCKFILE"

# Start a background worker
sleep 60 &
CHILD_PID=$!

# ... script body ...
echo "Working in $TMPDIR"

# cleanup() runs automatically on exit, error, or Ctrl+C
```

Always capture `$?` as the very first thing inside the cleanup function — subsequent commands will overwrite it. Then call `exit "$rc"` at the end to preserve the original exit code for callers.

The ERR trap — centralised error handling

ERR trap for logging failures with context

```
#!/usr/bin/env bash
set -euo pipefail

on_error() {
    local rc=$?
    local line=$1
    # BASH_COMMAND holds the command that failed
    printf '\033[31m[ERROR]\033[0m Line %d: command "%s" exited with code %d\n' \
        "$line" "$BASH_COMMAND" "$rc" >&2
    # Print a mini stack trace
    local i
    printf '  Stack:\n' >&2
    for (( i=1; i < ${#FUNCNAME[@]}; i++ )); do
        printf '    [%d] %s() at %s line %d\n' \
```

```
            "$i" "${FUNCNAME[$i]}" "${BASH_SOURCE[$i]}" "${BASH_LINENO[$((i-1))]}" >&2
    done
}
# Pass $LINENO as an argument so on_error knows where it was called
trap 'on_error $LINENO' ERR

inner() { false; }   # a function that fails
outer() { inner; }   # calls inner
outer
[ERROR] Line 18: command "false" exited with code 1
  Stack:
    [1] inner() at script.sh line 18
    [2] outer() at script.sh line 19
    [3] main script at script.sh line 20
```

The ERR trap does not fire inside functions where the error is tested (e.g. `if cmd; then` or `cmd || true`). It fires only when a command fails and the failure would propagate — the same condition that triggers `set -e`.

3 — Sending Signals: `kill`, `pkill`, `killall`

Sending signals to processes

```
# ── kill — send a signal to a PID or job spec ──────────────────
kill      1234          # send SIGTERM (default) to PID 1234
kill -TERM  1234        # same, explicit
kill -15    1234        # same, by number
kill -KILL  1234        # SIGKILL — last resort, unkillable by anything else
kill -HUP   1234        # SIGHUP — often means "reload config" for daemons
kill -0     1234        # test if process exists (no signal sent, just check)
kill -0     1234 2>/dev/null && echo "process is alive"

# Send to multiple PIDs
kill -TERM 1234 5678 9012

# Send to a job by job spec (only works in interactive shells or with set -m)
sleep 100 &
kill -TERM %1           # %1 = job 1
kill -TERM %sleep       # %sleep = job named sleep

# Send to a process group (negative PID = whole group)
kill -TERM -1234        # send SIGTERM to all processes in group 1234
kill -TERM -$$          # send SIGTERM to the current script's process group
```

```
# ── pkill — send signal by process name or attribute ──────────
pkill myapp              # SIGTERM all processes named "myapp"
pkill -KILL myapp        # force-kill
pkill -u $USER python3   # only kill python3 owned by current user
pkill -f 'worker.py --queue high'   # match full command line (not just name)
pkill -P $$              # kill all direct children of the current shell

# ── killall — kill by exact name (GNU/Linux) ──────────────────
killall firefox          # kill all processes named exactly "firefox"
killall -e myapp         # -e = exact match (don't truncate at 15 chars)
killall -w myapp         # -w = wait until all processes die

# ── Graceful terminate, then force if needed ──────────────────
kill_gracefully() {
    local pid=$1 timeout=${2:-10}
    kill -TERM "$pid" 2>/dev/null || return 0  # already gone
    local i
    for (( i=0; i < timeout; i++ )); do
        kill -0 "$pid" 2>/dev/null || return 0  # gone
        sleep 1
    done
    kill -KILL "$pid" 2>/dev/null || true
    echo "Process $pid did not stop gracefully; SIGKILL sent" >&2
}

kill_gracefully 1234 5   # give 5 seconds before SIGKILL
```

4 — Job Control: `bg`, `fg`, `jobs`, `disown`

Job control lets the shell manage multiple processes, suspending and resuming them as needed. It's enabled by default in interactive shells; in scripts, you need `set -m` to enable it. Each job has a job number (like `%1`) as well as a PID.

```
Job states:

  Running ──Ctrl+Z──▶ Stopped ──bg──▶ Running (background)
  Running (background) ──fg──▶ Running (foreground)
  Running (background) ──exits──▶ Done (reported at next prompt)

Job spec notation:
  %1          job number 1
  %%  or %+   most recently started/resumed job
  %-          second most recent job
```

```
%sleep      job whose command starts with "sleep"
%?log       job whose command contains "log"
```

Interactive job control

```
# Start a background job
sleep 100 &
[1] 4872

# List all jobs
jobs
[1]+  Running    sleep 100 &

jobs -l                  # show PIDs too
[1]+  4872 Running    sleep 100 &

jobs -p                  # show only PIDs

# Bring job to foreground (reclaims terminal)
fg %1
# Now press Ctrl+Z to suspend it
[1]+  Stopped    sleep 100

# Send it to background, still running
bg %1
[1]+ sleep 100 &

# ── disown — detach a job from the shell ──────────────────
# Without disown: when the shell exits, SIGHUP is sent to all jobs
long_running_process &
disown                   # disown %% (most recent job)
# Process now survives terminal close; shell no longer tracks it

disown %1               # disown specific job
disown -h %1            # disown but keep in jobs table (only removes SIGHUP)
disown -a               # disown all background jobs
```

Job control in scripts with `set -m`

Enabling job control in non-interactive scripts

```bash
#!/usr/bin/env bash
# Job control is off by default in scripts — enable it explicitly
set -m

sleep 30 &
JOB_PID=$!
JOB_NUM=$(jobs -l | awk '/'"$JOB_PID"'/{gsub(/[^0-9]/, "", $1); print $1}')

# With set -m, you can use job specs in kill/fg/bg
kill -STOP %1            # pause the job
kill -CONT %1            # resume it

# More commonly in scripts: just use PIDs directly
kill -TERM "$JOB_PID"

# Process groups: with set -m, each background job gets its own process group
# Kill the whole group (including any children the job spawned):
kill -TERM -"$JOB_PID"   # negative PID = process group
```

5 — `wait`: Collecting Background Jobs

`wait` pauses the script until background jobs complete, and crucially returns their exit codes. Without `wait`, background job failures are silently ignored.

wait, wait PID, and wait -n

```bash
# ── wait with no arguments — wait for ALL background jobs ──────
sleep 2 &
sleep 3 &
wait                     # blocks until both sleeps finish
echo "all done"

# ── wait PID — wait for a specific process ─────────────────────
sleep 5 &
pid=$!
wait "$pid"
rc=$?
echo "sleep exited with: $rc"

# ── Capturing exit codes from parallel jobs ────────────────────
do_work() {
```

```
    local id=$1
    echo "Worker $id starting"
    sleep $(( RANDOM % 3 + 1 ))
    (( id == 2 )) && return 1   # worker 2 deliberately fails
    echo "Worker $id done"
}

pids=()
for i in 1 2 3; do
    do_work "$i" &
    pids+=( $! )
done

failed=0
for pid in "${pids[@]}"; do
    if ! wait "$pid"; then
        echo "PID $pid failed" >&2
        (( failed++ ))
    fi
done
(( failed > 0 )) && { echo "$failed job(s) failed"; exit 1; }
echo "All workers succeeded"
```

`wait -n` — act as soon as any job finishes (bash 5.1+)

wait -n and wait -p — first-completed patterns

```
# wait -n returns when the FIRST background job finishes
# Returns that job's exit code

#!/usr/bin/env bash
# Requires bash 5.1+ for -p (store PID of the finished job)
set -euo pipefail

# Start several jobs
for url in \
    https://api.example.com/health \
    https://db.example.com/health \
    https://cache.example.com/health
do
    curl -sf "$url" > /dev/null &
done

# wait -n: wait for ANY one job to finish, capture its PID with -p
```

```
failed=0
while wait -n -p finished_pid 2>/dev/null; do
    true   # loop until all background jobs are done
done || (( failed++ ))
# Note: the loop exits when there are no more background jobs
# Return value of wait -n is the finished job's exit code

# Simpler pattern: bounded parallelism with wait -n
MAX_JOBS=4
running=0

for item in a b c d e f g h; do
    (( running >= MAX_JOBS )) && { wait -n; (( running-- )); }
    process_item "$item" &
    (( running++ ))
done
wait   # wait for remaining jobs
```

`wait -n` requires bash 4.3+. The `-p var` option to store the finished PID requires bash 5.1+. Check with `echo $BASH_VERSION`. For older bash, track PIDs in an array and poll with `kill -0`.

6 — Graceful Shutdown Patterns

A graceful shutdown means: stop accepting new work, finish what's in progress, clean up resources, and exit with an accurate status code. The challenge is that SIGTERM can arrive at any moment — you can't predict whether it hits a sleep, a curl, or a database commit. The patterns below handle this robustly.

Signal forwarding — don't leave children orphaned

Forwarding signals to child processes

```
#!/usr/bin/env bash
set -euo pipefail
# Problem: when the shell receives SIGTERM, child processes keep running
# unless you explicitly forward the signal

child_pid=""

forward_signal() {
    local sig=$1
    # Kill the child with the same signal we received
    [[ -n "$child_pid" ]] && kill -"$sig" "$child_pid" 2>/dev/null || true
```

```
}

trap 'forward_signal TERM' TERM
trap 'forward_signal INT'  INT
trap 'forward_signal HUP'  HUP

# Launch the real worker
./my_worker &
child_pid=$!

# Wait for it, but resume waiting if interrupted by a signal
# (trap handlers interrupt wait — the "|| true" re-enters the loop)
while ! wait "$child_pid"; do
    # wait was interrupted (signal arrived) — check if child is still alive
    kill -0 "$child_pid" 2>/dev/null || break
done
```

The shutdown flag pattern — controlled loop termination

Using a flag variable for clean loop exit

```
#!/usr/bin/env bash
set -euo pipefail

SHUTDOWN=0

on_shutdown() {
    echo "Shutdown signal received — finishing current item..." >&2
    SHUTDOWN=1
}
trap on_shutdown TERM INT HUP

# Main processing loop — check the flag at the top of each iteration
while (( SHUTDOWN == 0 )); do
    # Fetch next item from queue (this might block)
    item=$(fetch_next_item) || break

    # Process — we finish this even if shutdown was requested during fetch
    process_item "$item"
done

echo "Worker exited cleanly"

# ── SIGUSR1 for operational control ──────────────────────
```

```
# Use SIGUSR1/SIGUSR2 for custom actions like dumping stats or rotating logs
STATS_REQUESTED=0

dump_stats() {
    STATS_REQUESTED=1
}
trap dump_stats USR1

# In the main loop, check and handle the flag:
if (( STATS_REQUESTED )); then
    print_stats
    STATS_REQUESTED=0
fi
# Send stats from another terminal:  kill -USR1 <script_pid>
```

Daemon-style restart-on-exit wrapper

Supervising a child process with restart logic

```
#!/usr/bin/env bash
# supervisor.sh — restart a command if it exits unexpectedly
set -uo pipefail

CMD=( "$@" )              # the command to supervise
MAX_RESTARTS=${MAX_RESTARTS:-5}
RESTART_DELAY=${RESTART_DELAY:-3}
QUIT=0
restarts=0

trap 'QUIT=1' TERM INT HUP

while (( QUIT == 0 && restarts < MAX_RESTARTS )); do
    echo "[supervisor] Starting: ${CMD[*]}"

    "${CMD[@]}" &
    child=$!

    # Wait for child; resume after signal interruption
    while ! wait "$child" 2>/dev/null; do
        kill -0 "$child" 2>/dev/null || break
    done
    rc=$?

    (( QUIT )) && break     # we initiated the shutdown, don't restart
```

```
    (( restarts++ ))
    echo "[supervisor] Exited ($rc) — restart $restarts/$MAX_RESTARTS in ${RESTART_DELAY}s"
    sleep "$RESTART_DELAY"
done

(( restarts >= MAX_RESTARTS )) && echo "[supervisor] Max restarts reached. Giving up."
echo "[supervisor] Exiting"
```

7 — Trap Gotchas and Best Practices

Common trap mistakes and how to avoid them

```
# ── Gotcha 1: trap is NOT inherited by subshells ──────────────
trap 'echo "parent cleanup"' EXIT
(
    # This subshell does NOT inherit the EXIT trap — runs with no trap
    echo "in subshell"
)
# The parent's EXIT trap still fires when the parent exits

# ── Gotcha 2: trap is NOT inherited by child processes ────────
# A script called by your script starts with clean traps
# (unless you export with export -f or the child is a function)

# ── Gotcha 3: trap string vs function name timing ─────────────
file="initial"
# BAD: the string is evaluated NOW — $file captures "initial"
trap "rm -f $file"   EXIT   # ← double quotes: expands immediately

# GOOD: single quotes — $file evaluated when the trap fires
trap 'rm -f "$file"' EXIT   # ← single quotes: evaluates at trap time

# BEST: use a function — most readable and testable
my_cleanup() { rm -f "$file"; }
trap my_cleanup EXIT        # no quotes needed for function name

# ── Gotcha 4: stacking traps — each trap call REPLACES the previous ──
trap 'echo first'  EXIT
trap 'echo second' EXIT     # replaces the first trap!
# Only "second" will print — not "first"
```

```
# Fix: chain in a single trap, or use a cleanup function that accumulates
CLEANUP_TASKS=()
add_cleanup() { CLEANUP_TASKS+=( "$1" ); }
run_cleanups() {
    local rc=$?
    local task
    for task in "${CLEANUP_TASKS[@]}"; do
        eval "$task" || true   # run each, don't stop on failure
    done
    exit "$rc"
}
trap run_cleanups EXIT

# Now register individual cleanup actions at any point in the script
add_cleanup 'rm -f /tmp/myfile'
add_cleanup 'echo "Done cleaning up"'

# ── Gotcha 5: SIGKILL can never be caught ──────────────────
trap 'echo caught' KILL     # ERROR: cannot trap SIGKILL
# This is by design — SIGKILL is the OS's emergency stop
```

8 — Quick Reference

trap syntax

FORM	EFFECT
`trap 'cmd' SIG`	Run cmd when SIG is received
`trap fn SIG`	Call function fn when SIG is received
`trap 'cmd' SIG1 SIG2`	Same command for multiple signals
`trap '' SIG`	Ignore signal SIG
`trap - SIG`	Reset signal to default behaviour
`trap -p`	Print all active traps
`trap 'cmd' EXIT`	Run on any shell exit
`trap 'cmd' ERR`	Run when any command fails (non-zero exit)
`trap 'cmd' DEBUG`	Run before every simple command

Job control commands

COMMAND	EFFECT
`cmd &`	Run cmd in background; sets $! to its PID
`jobs`	List background jobs with job numbers
`jobs -l`	List jobs with PIDs
`fg %N`	Bring job N to foreground
`bg %N`	Resume stopped job N in background
`disown %N`	Detach job N (survives terminal close)
`disown -h %N`	Stop job receiving SIGHUP (but keep in jobs table)
`wait`	Wait for all background jobs
`wait $pid`	Wait for a specific PID; returns its exit code
`wait -n`	Wait for any one job to finish (bash 4.3+)
`wait -n -p var`	Wait for any job; store its PID in var (bash 5.1+)
`kill -0 $pid`	Test if process exists (no signal sent)
`pkill -P $$`	Kill all direct children of this shell

🖍️ Exercises

These exercises focus on writing scripts that behave correctly under interruption — the real test of signal handling. Run them in a terminal and test by pressing Ctrl+C or running `kill -TERM <pid>` from another window.

EXERCISE 1

Write a script called `safe_processor.sh` that reads lines from a file (passed as an argument), processes each one with a simulated 1-second operation, and — using a trap and shutdown flag — finishes the current item before exiting cleanly when it receives SIGTERM or SIGINT. It should print how many items it processed and whether it exited early.

Hint: set `SHUTDOWN=0` and trap TERM and INT to set it to 1. The `while` loop checks `(( SHUTDOWN == 0 ))`. Use `sleep 1` to simulate processing. The trap handler should not call `exit` directly — let the loop terminate naturally.

▶ Show Sample Solution

SAMPLE SOLUTION

```bash
#!/usr/bin/env bash
# safe_processor.sh  FILE
set -uo pipefail

FILE="${1:?Usage: safe_processor.sh FILE}"
[[ -f "$FILE" ]] || { echo "Error: '$FILE' not found" >&2; exit 1; }

SHUTDOWN=0
processed=0
total=$(wc -l < "$FILE")

on_signal() {
    echo ""
    echo "Signal received — finishing current item and shutting down..." >&2
    SHUTDOWN=1
}
trap on_signal TERM INT

echo "Processing $total items from $FILE (PID: $$)"
echo "Send SIGTERM with:  kill -TERM $$"

while (( SHUTDOWN == 0 )) && IFS= read -r line; do
    printf "  Processing: %s ... " "$line"
    sleep 1   # simulate work
    printf "done\n"
    (( processed++ ))
done < "$FILE"

# Report result
echo
if (( SHUTDOWN )); then
    printf "\033[33mEarly exit:\033[0m processed %d of %d items\n" \
        "$processed" "$total"
    exit 130     # conventional: 128 + signal number (SIGINT=2)
else
    printf "\033[32mComplete:\033[0m processed all %d items\n" "$processed"
fi
```

EXERCISE 2

Write a script called `parallel_jobs.sh` that takes a list of commands (one per line from stdin or a file) and runs them in parallel with a configurable concurrency limit (`MAX_JOBS` environment variable, default 3). It should collect each job's exit code, report which commands succeeded and which failed, and exit with a non-zero code if any job failed. Use `wait $pid` to collect exit codes.

Hint: maintain a `pids` array mapping PID to command string. Use `jobs -p | wc -l` or a counter variable to track running jobs. When the count reaches `MAX_JOBS`, call `wait -n` (or loop over pids checking `kill -0`) before launching the next. At the end, wait for all remaining PIDs.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# parallel_jobs.sh  [FILE]   (reads commands from FILE or stdin)
set -uo pipefail

MAX_JOBS=${MAX_JOBS:-3}

# pids[PID]="command string"
declare -A pids=()
declare -A failed_cmds=()
running=0

reap_one() {
    # Wait for any one job to finish; record failures
    local pid rc
    for pid in "${!pids[@]}"; do
        if ! kill -0 "$pid" 2>/dev/null; then
            # Process is gone — collect exit code
            wait "$pid" && rc=0 || rc=$?
            if (( rc != 0 )); then
                failed_cmds["$pid"]="${pids[$pid]} (exit $rc)"
                printf '\033[31m[FAIL]\033[0m %s\n' "${pids[$pid]}" >&2
            else
                printf '\033[32m[OK]\033[0m   %s\n' "${pids[$pid]}"
            fi
            unset 'pids[$pid]'
            (( running-- ))
            return
        fi
    done
```

```
    # All still running — sleep briefly and retry
    sleep 0.1
}

total=0

while IFS= read -r cmd; do
    [[ -z "$cmd" || "$cmd" == '#'* ]] && continue
    # Wait until a slot opens
    while (( running >= MAX_JOBS )); do
        reap_one
    done
    # Launch
    eval "$cmd" &
    pids[$!]="$cmd"
    (( running++ ))
    (( total++ ))
done

# Wait for remaining jobs
while (( running > 0 )); do
    reap_one
done

# Summary
nfailed=${#failed_cmds[@]}
printf '\n%d jobs total — %d failed\n' "$total" "$nfailed"
(( nfailed > 0 )) && exit 1 || exit 0

# Usage:
# echo -e "sleep 1\nsleeep 1\nsleep 2" | MAX_JOBS=2 ./parallel_jobs.sh
```

EXERCISE 3

Write a script called `monitored_run.sh` that runs a command passed as arguments, sets a timeout using `SIGALRM` or a background sleep-and-kill pattern, and produces a structured report: command, start time, end time, duration in seconds, exit code, and whether it was killed due to timeout. The timeout should be configurable via a `TIMEOUT` environment variable (default: 10 seconds).

Hint: the pure-bash timeout pattern is: start the command as a background job, also start a `sleep $TIMEOUT &` job, then `wait -n` for whichever finishes first — if the sleep wins, kill the command. Store both PIDs. The `SECONDS` built-in variable tracks elapsed time since the script started.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# monitored_run.sh  COMMAND [ARGS...]
set -uo pipefail

TIMEOUT=${TIMEOUT:-10}
(( $# > 0 )) || { echo "Usage: monitored_run.sh COMMAND [ARGS]" >&2; exit 1; }

cmd_str="$*"
start_time=$(date '+%Y-%m-%d %H:%M:%S')
start_sec=$SECONDS

# Start the target command
"$@" &
cmd_pid=$!

# Start the timeout watchdog
sleep "$TIMEOUT" &
sleep_pid=$!

timed_out=0
rc=0

# Wait for whichever finishes first
# We poll rather than use wait -n for broader compatibility
while true; do
    # Check if the command is done
    if ! kill -0 "$cmd_pid" 2>/dev/null; then
        kill "$sleep_pid" 2>/dev/null || true
        wait "$cmd_pid" && rc=0 || rc=$?
        break
    fi
    # Check if timeout elapsed
    if ! kill -0 "$sleep_pid" 2>/dev/null; then
        timed_out=1
        kill -TERM "$cmd_pid" 2>/dev/null || true
        sleep 1
        kill -KILL "$cmd_pid" 2>/dev/null || true
        rc=124   # GNU timeout convention
```

```
        break
    fi
    sleep 0.1
done

end_time=$(date '+%Y-%m-%d %H:%M:%S')
duration=$(( SECONDS - start_sec ))

# Structured report
printf '\n╔══ Run Report ═══════════════════════════\n'
printf '  Command   : %s\n'   "$cmd_str"
printf '  Start     : %s\n'   "$start_time"
printf '  End       : %s\n'   "$end_time"
printf '  Duration  : %ds\n'  "$duration"
printf '  Exit code : %d\n'   "$rc"
if (( timed_out )); then
    printf '  Timeout   : \033[31mYES — killed after %ds\033[0m\n' "$TIMEOUT"
else
    printf '  Timeout   : \033[32mno\033[0m\n'
fi
printf '╚═════════════════════════════════════════\n'

exit "$rc"
```

EXERCISE 4

Write a script called `cleanup_demo.sh` that demonstrates the stackable cleanup pattern from section 7. It should create a temp directory, a lock file, and a background process during its startup, registering each as a separate cleanup task using an `add_cleanup` function. It should then deliberately fail (using `false` or a bad command) to show that all cleanup tasks still run, in registration order. Print each cleanup step as it runs.

Hint: implement the `CLEANUP_TASKS` array and `run_cleanups` function from the gotchas section. Make the cleanup functions print a message like `[cleanup] removing /tmp/demo.XYZ` before doing the actual removal. Use `set -e` so the deliberate failure triggers the EXIT trap.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# cleanup_demo.sh — stackable cleanup pattern
```

```bash
set -euo pipefail

# ── Cleanup stack ──────────────────────────────────────────
CLEANUP_TASKS=()

add_cleanup() {
    CLEANUP_TASKS+=( "$1" )
}

run_cleanups() {
    local rc=$?
    echo ""
    printf '\033[36m[cleanup]\033[0m Running %d cleanup tasks...\n' \
        "${#CLEANUP_TASKS[@]}" >&2
    local task
    for task in "${CLEANUP_TASKS[@]}"; do
        eval "$task" || printf '\033[31m[cleanup]\033[0m Task failed: %s\n' \
            "$task" >&2
    done
    printf '\033[36m[cleanup]\033[0m Done. Original exit code: %d\n' \
        "$rc" >&2
    exit "$rc"
}

trap run_cleanups EXIT

# ── Startup: acquire resources ─────────────────────────────
echo "[1] Creating temp directory..."
TMPDIR=$(mktemp -d)
echo "    Created: $TMPDIR"
add_cleanup "printf '[cleanup] removing temp dir %s\n' \"$TMPDIR\" >&2; rm -rf \"$T

echo "[2] Creating lock file..."
LOCKFILE=/tmp/cleanup_demo_$$.lock
touch "$LOCKFILE"
echo "    Created: $LOCKFILE"
add_cleanup "printf '[cleanup] releasing lock %s\n' \"$LOCKFILE\" >&2; rm -f \"$LOC

echo "[3] Starting background process..."
sleep 60 &
BG_PID=$!
echo "    PID: $BG_PID"
add_cleanup "printf '[cleanup] stopping background PID %d\n' $BG_PID >&2; kill $BG_
```

```
# ── Deliberate failure ──────────────────────────────────
echo
echo "[!] About to fail deliberately..."
false   # set -e causes EXIT trap to fire here

# This line never runs
echo "This should not print"
```

🌀 Intermediate Topic 4 — Subshells and Execution Environments

One of the most common sources of baffling bugs in bash scripts is the variable that "disappears" — a counter that never increments, a flag that's always unset after a loop, an array that's empty when it should have elements. The culprit, almost every time, is an unexpected subshell. Understanding exactly when bash creates a subshell, what it copies into it, what it doesn't, and how information can (and can't) flow back to the parent is the key to writing scripts that behave as intended. This chapter maps the entire execution environment model.

1 — The Execution Environment

Every running bash shell has an *execution environment*: the complete set of state that determines what commands do. When bash creates a subshell it uses `fork()` — the OS creates an exact copy of the current process. The subshell starts with an identical environment but then diverges: any changes it makes are local to its copy and never flow back to the parent.

```
The execution environment — what each shell instance holds:

  Variables      — shell variables (only exported ones pass to child processes)
  Functions      — defined with fn() { } or function fn { }
  Aliases        — defined with alias
  Options        — set -e, set -u, set -x, shopt -s extglob, etc.
  Traps          — registered with trap (NOT inherited by subshells)
  Working dir    — current directory (cd in a subshell stays local)
  Umask          — file creation mask
  Open FDs       — file descriptors (inherited, but exec in subshell stays local)
  $0, $1…        — positional parameters
  $$             — PID (subshell keeps the parent's PID — use $BASHPID for own PID)

What a subshell gets (fork copy):   Everything above, at the moment of fork
What flows back to the parent:      Nothing — ever. Only the exit code.
Exception — exporting via stdout:   capture output with $() and re-assign in parent
```

Proving the isolation — variables don't propagate back

```
x=10

# Subshell created by ( )
(
```

```
    x=99
    echo "inside subshell: x=$x"   → inside subshell: x=99
)
echo "after subshell:  x=$x"     → after subshell:  x=10   ← unchanged

# Same for cd — directory changes don't escape a subshell
echo "before: $(pwd)"
(
    cd /tmp
    echo "inside: $(pwd)"          → /tmp
)
echo "after:  $(pwd)"              → /original/dir  ← unchanged

# $$ vs $BASHPID
echo "parent PID via \$\$:      $$"
echo "parent PID via \$BASHPID: $BASHPID"
(
    # $$ still shows the PARENT pid (historical bash quirk)
    # $BASHPID shows the actual subshell PID
    echo "subshell \$\$:      $$"      → same as parent (quirk!)
    echo "subshell \$BASHPID: $BASHPID" → different PID (correct)
)

# $BASH_SUBSHELL — depth counter (0 = top-level, increments with nesting)
echo "depth: $BASH_SUBSHELL"       → 0
(
    echo "depth: $BASH_SUBSHELL"   → 1
    (
        echo "depth: $BASH_SUBSHELL" → 2
    )
)
```

2 — What Creates a Subshell?

Many constructs create subshells silently. Knowing them all is essential to predicting where your variables will and won't be visible.

CONSTRUCT	SUBSHELL?	NOTES
`( commands )`	**Yes**	Explicit subshell group — always a fork
`{ commands; }`	**No**	Brace group — runs in the *current* shell
`$(commands)`	**Yes**	Command substitution — output captured, variables lost

CONSTRUCT	SUBSHELL?	NOTES
`cmd1 | cmd2`	Yes	Both sides of a pipe run in subshells (by default)
`cmd &`	Yes	Background job — fork + runs in background
`<(cmd)`	Yes	Process substitution — cmd runs in a subshell
`>(cmd)`	Yes	Output process substitution
`bash script.sh`	Yes	New process entirely (child process, not just subshell)
`source script.sh` / `. script.sh`	No	Runs in the current shell — variables persist
`exec cmd`	No	Replaces the current shell — no fork
`if cmd; then` / `while cmd`	No	The compound command itself runs in current shell
`cmd1 && cmd2` / `cmd1 || cmd2`	No	Both sides run in current shell

The pipeline subshell rule: In bash (unlike ksh/zsh), *every* command in a pipeline — including the last one — runs in a subshell. This means a `while read` loop at the end of a pipe cannot modify variables in the parent script. This is the single most common "my variable is always zero" bug.

The last-pipe option — `shopt -s lastpipe`

lastpipe — run the last pipeline command in the current shell

```
# ── The classic bug ──────────────────────────────────────
count=0
cat data.txt | while IFS= read -r line; do
    (( count++ ))
done
echo "count: $count"   → 0   (always zero — loop ran in subshell)

# ── Fix 1: process substitution (Topic 2 pattern) ────────
count=0
while IFS= read -r line; do
    (( count++ ))
done < <(cat data.txt)
echo "count: $count"   → correct count (loop in current shell)

# ── Fix 2: shopt -s lastpipe (bash 4.2+, non-interactive only) ─
shopt -s lastpipe
count=0
cat data.txt | while IFS= read -r line; do
    (( count++ ))
```

```
done
echo "count: $count"   → correct (lastpipe runs last cmd in current shell)
# Note: lastpipe only works when job control is disabled (set +m),
# which is the default in non-interactive scripts

# ── Fix 3: redirect a file directly (no pipe, no subshell) ──────
count=0
while IFS= read -r line; do
    (( count++ ))
done < data.txt
echo "count: $count"   → correct (no subprocess at all)
```

3 — ( ) vs { } — Subshell vs Brace Group

These two constructs look similar and do similar things — but one creates a subprocess, the other doesn't. Knowing which to reach for is one of the most impactful micro-decisions in bash scripting.

( ) — SUBSHELL GROUP — NEW PROCESS

```
# Creates a child process via fork()
# Variables, cd, set options — all isolated
# Traps are NOT inherited
# More expensive (fork + exec overhead)
# No semicolon needed before closing )

(
    cd /tmp          # stays in subshell
    set -e           # only affects subshell
    x=99             # invisible to parent
    risky_cmd        # if it fails, only subs
)
```

{ } — BRACE GROUP — SAME PROCESS

```
# No fork — runs in the current shell
# Variables and cd persist to parent
# Traps ARE shared
# Fast (no subprocess overhead)
# REQUIRES semicolon before closing }

{
    cd /tmp          # changes current shell
    set -e           # affects the whole scr
    x=99             # visible after the blo
    risky_cmd        # failure propagates ou
}
```

When to use each deliberately

```
# ── Use ( ) when you WANT isolation ──────────────────────────

# Safe cd: change directory, do work, return automatically
( cd /path/to/project && make )
# No risk of stranding the parent in /path/to/project
```

```bash
# Try a risky operation without affecting the script's set -e
if ( ssh -o ConnectTimeout=3 "$host" 'exit 0' ); then
    echo "$host is reachable"
fi

# Modify options or traps without affecting the outer script
(
    set +e          # temporarily disable exit-on-error
    some_flaky_cmd
    rc=$?
    # set -e is restored automatically when subshell exits
)
# Back to set -e here

# ── Use { } when you want grouping WITHOUT isolation ──────────

# Group redirections over multiple commands without a subshell
{
    date
    uptime
    df -h
} > system_report.txt

# Short-circuit with || and still affect parent state
{ echo "Error: missing argument" >&2; exit 1; }

# Group a multi-line pipeline that needs to set a variable
{
    echo "start"
    process_data
    echo "end"
} | tee output.log
# Note: brace group itself is still subshelled when it's in a pipeline!
```

Even a `{ }` brace group runs in a subshell if it appears inside a pipeline (e.g. `{ ...; } | cmd`). The pipeline is what creates the subshell, not the braces. Braces only prevent an extra fork that `( )` would add.

4 — Environment Variables and Inheritance

There are two kinds of variables in bash: *shell variables* (local to the current shell) and *environment variables* (inherited by child processes via the process's `environ`). The distinction matters whenever you launch an external command, a script, or a subshell.

export, env, and inheritance rules

```bash
# Shell variable — NOT inherited by child processes
local_var="hello"
bash -c 'echo "$local_var"'      → (empty — child doesn't see it)

# Exported variable — IS inherited
export ENV_VAR="world"
bash -c 'echo "$ENV_VAR"'         → world

# export makes an existing variable part of the environment
my_var="value"
export my_var                    # promote to environment

# Inline export for one command only — does not persist
API_KEY="secret" curl https://api.example.com
# API_KEY is only in curl's environment, not the shell

# Functions are NOT inherited by child processes unless exported
greet() { echo "Hello, $1"; }
bash -c 'greet "world"'           → greet: command not found

export -f greet                  # export the function
bash -c 'greet "world"'           → Hello, world

# unexport — remove from environment (keep as shell variable)
export -n my_var                 # demote from environment

# env — run command with a clean or modified environment
env -i bash -c 'echo "$HOME"'    → (empty — -i clears all env vars)
env -i HOME=/tmp PATH=/bin bash # start with minimal env

# Inspect the environment
printenv                         # list all exported variables
printenv PATH                    # print a specific variable
declare -x                       # same as printenv but in declare format
```

5 — source vs Execute: Loading vs Running

When you run a script with `bash script.sh` or `./script.sh`, bash forks a new process. When you *source* a script with `source script.sh` or `. script.sh`, the commands run directly in the current shell — no fork. This distinction determines whether the script's variables, functions, and directory changes affect the calling environment.

BASH SCRIPT.SH — NEW PROCESS

```
# Completely separate process
# Inherits exported variables only
# Its functions/vars stay inside it
# Its cd doesn't move the parent
# Parent gets exit code only

# script.sh:
MY_FUNC="defined"
cd /tmp

# After running it:
echo "$MY_FUNC"  → (empty)
pwd              → original dir
```

SOURCE SCRIPT.SH — CURRENT SHELL

```
# No fork — runs in THIS shell
# All variables become available
# Functions are defined here
# cd actually changes your dir
# exit in sourced file exits YOU

# After sourcing it:
echo "$MY_FUNC"  → defined
pwd              → /tmp
```

Practical sourcing patterns

```
# ── Library files — source to load functions ──────────────
# lib/logging.sh defines log(), die(), warn() etc.
source "$(dirname "$0")/lib/logging.sh"
# or with dot syntax (POSIX-compatible alias):
. "$(dirname "$0")/lib/logging.sh"

# ── Config files — source to set variables ────────────────
# config.env contains:  DB_HOST=prod.db.example.com
[[ -f config.env ]] && source config.env

# ── Guard against sourcing vs executing ───────────────────
# Detect if the script is being sourced or executed directly
if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then
    # Running as a standalone script
    main "$@"
else
    # Being sourced as a library — only define functions, don't run
    true
fi
# This is the standard "library with optional main" pattern

# ── BASH_SOURCE array — stack of sourced files ────────────
# ${BASH_SOURCE[0]} = current file (or "" if interactive)
# ${BASH_SOURCE[1]} = the file that sourced us
# ${BASH_SOURCE[-1]} = the original script
```

```
where_am_i() {
    echo "Called from: ${BASH_SOURCE[1]}:${BASH_LINENO[0]}"
    echo "Defined in:  ${BASH_SOURCE[0]}"
}

# Robust path to current script (works whether sourced or executed)
SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)
```

Never use `exit` in a file intended to be sourced — it will exit the calling shell. Use `return` instead; it exits the sourced context only, just like returning from a function. This is a common gotcha in library scripts.

6 — exec: Replacing the Current Process

`exec` with a command replaces the running shell with the specified program — no fork, no parent left behind. It's used to hand off control to a final command (saving a process slot), to wrap scripts around another program, and — when used without a command — to set persistent redirections (covered in Topic 2).

exec patterns for process replacement

```
# ── Hand off to the real program ──────────────────────────
#!/usr/bin/env bash
# wrapper.sh — set up environment then become the real program
export APP_CONFIG=/etc/myapp/config.yaml
export LOG_LEVEL="INFO"
ulimit -n 65536        # raise file descriptor limit

exec /opt/myapp/bin/myapp "$@"
# This shell is GONE — myapp IS the process now
# The PID stays the same; myapp gets it
# Nothing after exec ever runs

# ── Use exec to save a process slot in pid-1 containers ──────
# In Docker, PID 1 receives signals. If your entrypoint is a shell
# and it calls exec, the app becomes PID 1 and gets signals directly
#!/bin/sh
# docker-entrypoint.sh
setup_config
exec "$@"   # exec whatever was passed as the docker CMD

# ── exec in a subshell — exits the subshell, not the parent ──
(
    exec cat /etc/hostname
```

```
    # exec replaces the subshell process with cat
    # parent shell is unaffected
)
echo "parent still running"

# ── exec for privilege separation ──────────────────────
# Drop privileges before exec'ing the app
if [[ $EUID -eq 0 ]]; then
    exec su -s /bin/bash -c "exec \"$0\" \"$@\"" appuser
fi
# Script re-runs itself as appuser; this branch only fires as root
```

7 — Returning Data from Subshells

Since variables can't propagate from a subshell to its parent, you need an explicit channel. There are four patterns, each with different trade-offs.

The four patterns for getting data out of a subshell

```
# ── Pattern 1: stdout capture with $() ─────────────────
# Clean, composable, but creates a subshell and only returns a string
get_hostname() {
    hostname -f
}
host=$(get_hostname)

# Limitation: trailing newlines are stripped by $()
with_newlines=$'line1\nline2\n'
captured="$(echo "$with_newlines")"  # trailing \n stripped!
# Workaround: add a sentinel, strip it after
captured="$(echo "$with_newlines"; echo x)"
captured="${captured%x}"             # remove sentinel

# ── Pattern 2: exit code only ──────────────────────────
# When you only need a pass/fail result — most efficient
if ( grep -q 'error' logfile.txt ); then
    echo "Errors found"
fi

# ── Pattern 3: temp file ───────────────────────────────
# For large or structured data; allows multiple outputs
tmpf=$(mktemp)
```

```
trap 'rm -f "$tmpf"' EXIT

(
    result1="computed"
    result2="also computed"
    declare -p result1 result2    # write declarations to the file
) > "$tmpf"
source "$tmpf"                    # reload the declarations in parent
echo "$result1  $result2"         → computed  also computed

# ── Pattern 4: namerefs (preferred for functions) ──────────────
# From Topic 1 — pass a variable name, write via nameref
# No subshell created — works inside functions in the same shell
compute() {
    local -n _out="$1"
    _out="the result"
}
compute my_var
echo "$my_var"                    → the result
# Note: functions called as $() STILL run in a subshell
# val=$(compute result)  ← nameref useless here, $() makes a subshell
```

8 — Performance, Gotchas, and Best Practices

Avoiding unnecessary subshells — the performance angle

```
# Every $() creates a fork. In a tight loop, this adds up.
# Benchmark on a modern Linux box: ~1–2ms per fork

# ── SLOW: $() in a loop ────────────────────────────────────────
for i in {1..100}; do
    ts=$(date '+%s')          # fork + exec date, 100 times
    echo "$i: $ts"
done

# ── FAST: use printf %(%s)T (bash built-in, no fork) ──────────
for i in {1..100}; do
    printf -v ts '%(%s)T' -1   # bash built-in timestamp, no fork
    echo "$i: $ts"
done

# ── Replace common $() patterns with built-ins ─────────────────
```

```
# dirname / basename — use parameter expansion instead
path=/usr/local/bin/myprog
dir="$(dirname "$path")"         # fork
dir="${path%/*}"                  # no fork — much faster
base="$(basename "$path")"       # fork
base="${path##*/}"                # no fork

# tr for case conversion — use ${var^^} / ${var,,} instead
upper="$(echo "$str" | tr '[:lower:]' '[:upper:]')"  # 2 forks
upper="${str^^}"                  # no fork

# wc -c for string length — use ${#var}
len="$(echo -n "$str" | wc -c)"  # 2 forks
len="${#str}"                     # no fork

# printf -v — assign to variable without a subshell
result="$(printf '%05d' $n)"     # fork for $() even though printf is builtin
printf -v result '%05d' "$n"     # no fork — assigns directly
```

Common gotchas checklist

```
# ── Gotcha 1: exit in a subshell only exits the subshell ──────
function check() {
    (
        [[ -f "$1" ]] || exit 1    # exits the () subshell, not the function
        echo "file exists"
    )
}
check /nonexistent
echo "still running"            → still running  (script continues!)
# Fix: check the subshell's exit code with $?

# ── Gotcha 2: set -e doesn't propagate into $() ──────────────
set -e
result=$(false; echo "still ran")
# 'false' fails but set -e doesn't kill the script here
# The $() subshell exits non-zero, and THAT propagates up
# So the assignment fails and set -e fires — but the echo ran first

# ── Gotcha 3: traps reset in subshells ────────────────────────
trap 'echo "parent cleanup"' EXIT
(
    trap -p EXIT    # shows empty — EXIT trap is NOT inherited by subshells
```

```
)

# ── Gotcha 4: read after a pipe ──────────────────────────
# (the classic bug — covered in section 2)
declare -a lines
cat file.txt | mapfile -t lines
echo "${#lines[@]}"  → 0  (mapfile ran in a subshell)

# Fix:
mapfile -t lines < file.txt       # no pipe, no subshell
# or:
mapfile -t lines < <(generate_lines)  # process substitution
```

9 — Quick Reference

Construct comparison

CONSTRUCT	SUBSHELL?	VARIABLES PERSIST?	CD PERSISTS?	TRAPS INHERITED?
`( cmds )`	Yes	No	No	No
`{ cmds; }`	No	Yes	Yes	Yes
`$(cmds)`	Yes	No	No	No
`cmd1 | cmd2`	Both sides	No	No	No
`cmd &`	Yes	No	No	No
`<(cmd)`	Yes	No	No	No
`source f` / `. f`	No	Yes	Yes	Yes
`bash f`	New process	No	No	No
`exec cmd`	Replaces shell	*n/a*	*n/a*	*n/a*

Key special variables for execution context

VARIABLE	CONTENT
`$$`	PID of the original shell (same value in subshells — a quirk)
`$BASHPID`	PID of the current shell/subshell (correct value)

VARIABLE	CONTENT
`$BASH_SUBSHELL`	Subshell nesting depth (0 = top level)
`${BASH_SOURCE[0]}`	Path of the current file (or empty if interactive)
`${BASH_SOURCE[-1]}`	Path of the outermost script
`${FUNCNAME[0]}`	Name of the current function (or "main")
`${BASH_LINENO[0]}`	Line number where the current function was called from

🖍️ Exercises

Each exercise is designed to expose a real-world consequence of subshell behaviour — the kinds of bugs that quietly ship to production. Trace through each one mentally before running it, then verify your predictions.

EXERCISE 1

The following script has a silent bug: the `total` variable is always 0 at the end. Identify exactly why, then rewrite it in three different correct ways — using process substitution, using `shopt -s lastpipe`, and using direct file redirection. All three versions must produce the correct sum.

```
total=0
seq 1 10 | while IFS= read -r n; do
    (( total += n ))
done
echo "Total: $total"   → Total: 0  (should be 55)
```

▶ Show Sample Solution

SAMPLE SOLUTION

```
# The bug: the pipe creates a subshell for the while loop.
# $total is incremented inside the subshell's copy, never the parent's.
# When the subshell exits, its copy of $total vanishes.

# ── Fix 1: process substitution — while loop runs in current shell
total=0
while IFS= read -r n; do
    (( total += n ))
done < <(seq 1 10)
echo "Total: $total"   → Total: 55
```

```
# ── Fix 2: shopt -s lastpipe — last cmd in pipe runs in current shell
shopt -s lastpipe
total=0
seq 1 10 | while IFS= read -r n; do
    (( total += n ))
done
echo "Total: $total"   → Total: 55
shopt -u lastpipe      # restore default if needed

# ── Fix 3: direct file redirection — no subprocess at all
total=0
tmpf=$(mktemp)
seq 1 10 > "$tmpf"
while IFS= read -r n; do
    (( total += n ))
done < "$tmpf"
rm -f "$tmpf"
echo "Total: $total"   → Total: 55

# Bonus: the cleanest one-liner (awk, no bash variable involved at all)
seq 1 10 | awk '{s+=$1} END{print "Total:", s}'  → Total: 55
```

EXERCISE 2

Write a script called `safe_cd.sh` that implements a `with_dir()` function which: changes to a specified directory, runs a command, then automatically returns to the original directory — even if the command fails. The function should work correctly whether it's a single command or a pipeline. Demonstrate it with three calls: a successful command, a failing command (verify the original directory is restored), and a command that uses the changed directory for a glob pattern.

Hint: the simplest implementation wraps the work in a subshell `( cd "$dir" && "$@" )`. The `cd` is local to the subshell. For pipelines, use `bash -c "..."` or a function with `eval`. Think about how to handle a command that needs to be a pipeline.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# safe_cd.sh
set -uo pipefail
```

```bash
with_dir() {
    # Usage: with_dir DIRECTORY COMMAND [ARGS...]
    local dir="${1:?with_dir: directory required}"
    shift
    [[ -d "$dir" ]] || { echo "with_dir: '$dir' is not a directory" >&2; return 1;
    # Subshell: cd is local, command runs, shell exits, parent is unaffected
    (
        cd "$dir" || return 1
        "$@"
    )
}

with_dir_eval() {
    # Variant that accepts a pipeline string (uses bash -c)
    local dir="${1:?with_dir_eval: directory required}"
    local cmd="$2"
    [[ -d "$dir" ]] || { echo "with_dir_eval: '$dir' is not a directory" >&2; retur
    ( cd "$dir" && eval "$cmd" )
}

# ── Demonstration ──────────────────────────────────────────
echo "Starting directory: $(pwd)"

# 1. Successful command
echo "\n--- Test 1: successful command ---"
with_dir /etc ls -1 hosts passwd
echo "After: $(pwd)"      → original dir (unchanged)

# 2. Failing command — directory must still be restored
echo "\n--- Test 2: failing command ---"
with_dir /tmp ls /this/does/not/exist || echo "(command failed, as expected)"
echo "After: $(pwd)"      → original dir (still unchanged)

# 3. Command using a glob in the changed directory
echo "\n--- Test 3: glob in changed directory ---"
with_dir /etc bash -c 'echo *.conf | tr " " "\n" | head -5'
echo "After: $(pwd)"      → original dir

# 4. Pipeline variant
echo "\n--- Test 4: pipeline ---"
with_dir_eval /var/log 'ls -lt | head -3'
echo "After: $(pwd)"      → original dir
```

EXERCISE 3

Write a library file called `lib_utils.sh` that: (1) can be both sourced and executed, (2) defines three utility functions: `trim()`, `is_number()`, and `repeat()`, (3) when executed directly (not sourced), runs a self-test that exercises each function and reports pass/fail, and (4) uses `${BASH_SOURCE[0]}` to detect which mode it's in. When sourced, it should define the functions silently. When executed, it should print a test report.

Hint: the pattern is `if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then run_tests; fi`. For `trim()`, use parameter expansion from Topic 1. For `is_number()`, use a regex with `[[ =~ ]]`. For `repeat()`, use a loop or printf. Write the test function to call each and compare actual to expected output.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# lib_utils.sh — utility library (sourceable and self-testing)

# ── Library functions ──────────────────────────────────

trim() {
    # Trim leading and trailing whitespace from a string
    # Usage: trim STRING  or  result=$(trim STRING)
    local str="$1"
    # Strip leading whitespace
    str="${str#"${str%%[! $'\t']*}"}"
    # Strip trailing whitespace
    str="${str%"${str##*[! $'\t']}"}"
    printf '%s' "$str"
}

is_number() {
    # Return 0 (true) if the argument is an integer or float
    # Usage: is_number VALUE
    [[ "$1" =~ ^-?[0-9]+(\.[0-9]+)?$ ]]
}

repeat() {
    # Repeat a string N times
    # Usage: repeat STRING COUNT
    local str="$1" n="${2:?repeat: count required}"
```

```
    local i
    for (( i=0; i < n; i++ )); do
        printf '%s' "$str"
    done
    printf '\n'
}

# ── Self-test (only runs when executed directly) ──────────────
_run_tests() {
    local pass=0 fail=0

    _assert_eq() {
        local desc="$1" got="$2" want="$3"
        if [[ "$got" == "$want" ]]; then
            printf '\033[32m  PASS\033[0m %s\n' "$desc"; (( pass++ ))
        else
            printf '\033[31m  FAIL\033[0m %s\n       got:  [%s]\n       want: [%s]\
                "$desc" "$got" "$want"; (( fail++ ))
        fi
    }
    _assert_true() {
        local desc="$1"; shift
        if "$@"; then
            printf '\033[32m  PASS\033[0m %s\n' "$desc"; (( pass++ ))
        else
            printf '\033[31m  FAIL\033[0m %s (returned false)\n' "$desc"; (( fail++
        fi
    }

    printf '\033[1mlib_utils.sh — self test\033[0m\n'
    printf '%.0s─' {1..40}; echo

    # trim()
    _assert_eq  "trim: leading spaces"        "$(trim '   hello')"          "hello"
    _assert_eq  "trim: trailing spaces"       "$(trim 'hello   ')"          "hello"
    _assert_eq  "trim: both sides"           "$(trim '  hello world  ')"   "hello w
    _assert_eq  "trim: already trimmed"      "$(trim 'no spaces')"         "no spac
    _assert_eq  "trim: empty string"         "$(trim '')"                  ""

    # is_number()
    _assert_true "is_number: integer"         is_number 42
    _assert_true "is_number: negative"        is_number -7
    _assert_true "is_number: float"           is_number 3.14
```

```
    _assert_true "is_number: rejects word"    ! is_number "abc"
    _assert_true "is_number: rejects mixed"   ! is_number "12abc"

    # repeat()
    _assert_eq  "repeat: 3 times"           "$(repeat 'ab' 3)"           "ababab"
    _assert_eq  "repeat: 0 times"           "$(repeat 'x' 0)"            ""
    _assert_eq  "repeat: 1 time"            "$(repeat '-' 1)"            "-"

    printf '%.0s─' {1..40}; echo
    printf '%d passed  %d failed\n' "$pass" "$fail"
    (( fail == 0 ))
}

# ── Entry point detection ──────────────────────────────────
if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then
    # Executed directly: run self-test
    _run_tests
fi
# When sourced: functions are silently defined, no tests run
```

EXERCISE 4

Write a script called `env_sandbox.sh` that runs a command in a controlled environment: it accepts a list of `KEY=VALUE` overrides as arguments followed by a `--` separator and then the command. It should run the command with: (a) the parent's environment as a base, (b) only specified overrides applied, and (c) a set of variables always removed (configurable via `SANDBOX_STRIP`, defaulting to `AWS_SECRET_ACCESS_KEY:GITHUB_TOKEN:DATABASE_URL`). Print the final environment and exit code. Use a subshell for isolation.

Hint: parse arguments until you hit `--`, collecting `KEY=VALUE` pairs into an array. Use `env -u KEY` to unset specific variables. The command to run is everything after `--`. Use `env KEY=VAL ... CMD` to set overrides and unset sensitive vars in one call.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# env_sandbox.sh  [KEY=VAL ...]  --  COMMAND [ARGS...]
set -uo pipefail

SANDBOX_STRIP=${SANDBOX_STRIP:-"AWS_SECRET_ACCESS_KEY:GITHUB_TOKEN:DATABASE_URL"}
```

```
# Parse: collect overrides until --, then the command
overrides=()
cmd=()
found_sep=0

for arg in "$@"; do
    if (( found_sep )); then
        cmd+=( "$arg" )
    elif [[ "$arg" == "--" ]]; then
        found_sep=1
    elif [[ "$arg" == *"="* ]]; then
        overrides+=( "$arg" )
    else
        echo "Usage: env_sandbox.sh [KEY=VAL ...] -- COMMAND [ARGS]" >&2
        exit 1
    fi
done

(( ${#cmd[@]} > 0 )) || { echo "No command specified after --" >&2; exit 1; }

# Build env -u flags for stripped variables
unset_flags=()
IFS=':' read -r -a strip_vars <<< "$SANDBOX_STRIP"
for v in "${strip_vars[@]}"; do
    unset_flags+=( -u "$v" )
done

# Print sandbox summary
printf '\033[36m[sandbox]\033[0m Command:   %s\n'  "${cmd[*]}"
printf '\033[36m[sandbox]\033[0m Overrides: %s\n'  "${overrides[*]:-none}"
printf '\033[36m[sandbox]\033[0m Stripped:  %s\n'  "$SANDBOX_STRIP"
echo

# Run in a subshell — isolation is belt-and-suspenders here
(
    env "${unset_flags[@]}" "${overrides[@]}" "${cmd[@]}"
)
rc=$?

printf '\n\033[36m[sandbox]\033[0m Exit code: %d\n' "$rc"
exit "$rc"
```

```
# Example usage:
# DATABASE_URL=postgres://prod ./env_sandbox.sh  DATABASE_URL=postgres://test  --
# GITHUB_TOKEN=secret ./env_sandbox.sh  --  env | grep GITHUB   ← GITHUB_TOKEN is s
```

🕐 Intermediate Topic 5 — Working with Dates and Times

Date and time handling trips up shell scripters more than almost any other topic. The tools are powerful but the interfaces are inconsistent — GNU `date` and BSD `date` (macOS) have completely different syntax for date arithmetic. Epoch seconds are the lingua franca that makes everything else predictable. This chapter covers every form of date formatting, arithmetic entirely in shell integers, timezone handling, comparing and aging timestamps, the bash built-in time formatting that avoids forks in tight loops, and scheduling decision patterns used in real-world cron jobs and maintenance scripts.

1 — The `date` Command and Format Strings

`date` with a `+FORMAT` argument controls its output. Every format directive starts with `%`. The result is a string you compose exactly as needed — for log files, filenames, display, or further arithmetic.

DIRECTIVE	MEANING	EXAMPLE OUTPUT
`%Y`	4-digit year	`2026`
`%y`	2-digit year	`26`
`%m`	Month (01–12)	`06`
`%d`	Day of month (01–31)	`09`
`%j`	Day of year (001–366)	`160`
`%H`	Hour, 24h (00–23)	`14`
`%I`	Hour, 12h (01–12)	`02`
`%M`	Minute (00–59)	`30`
`%S`	Second (00–60)	`05`
`%N`	Nanoseconds *(GNU only)*	`123456789`
`%s`	Unix epoch seconds since 1970-01-01 UTC	`1749470405`
`%A`	Full weekday name	`Monday`
`%a`	Abbreviated weekday	`Mon`
`%B`	Full month name	`June`
`%b`	Abbreviated month name	`Jun`

DIRECTIVE	MEANING	EXAMPLE OUTPUT
`%u`	Day of week (1=Mon … 7=Sun)	`1`
`%w`	Day of week (0=Sun … 6=Sat)	`1`
`%W`	Week number of year (Mon-based)	`23`
`%Z`	Timezone abbreviation	`UTC`
`%z`	UTC offset	`+0100`
`%:z`	UTC offset with colon *(GNU only)*	`+01:00`
`%F`	ISO date shorthand: `%Y-%m-%d`	`2026-06-09`
`%T`	Time shorthand: `%H:%M:%S`	`14:30:05`
`%R`	Hour:minute: `%H:%M`	`14:30`
`%D`	US date: `%m/%d/%y`	`06/09/26`
`%%`	Literal percent sign	`%`

Common date format recipes

```
# ── Formats for filenames (no spaces or colons) ──────────────
date '+%Y%m%d'              → 20260609
date '+%Y%m%d_%H%M%S'       → 20260609_143005
date '+%Y-%m-%dT%H%M%S'     → 2026-06-09T143005

# ── ISO 8601 — the universal machine-readable format ─────────
date '+%Y-%m-%d'            → 2026-06-09
date '+%Y-%m-%dT%H:%M:%S'   → 2026-06-09T14:30:05
date -Is                    → 2026-06-09T14:30:05+01:00  (GNU shorthand)
date '+%Y-%m-%dT%H:%M:%S%z' → 2026-06-09T14:30:05+0100

# ── Human-readable log timestamps ────────────────────────────
date '+%a %b %d %T %Z %Y'   → Mon Jun 09 14:30:05 UTC 2026
date '+[%Y-%m-%d %H:%M:%S]' → [2026-06-09 14:30:05]

# ── Epoch seconds — the key to all arithmetic ────────────────
date '+%s'                  → 1749470405

# ── Suppress newline / assign to variable ────────────────────
ts=$(date '+%Y%m%d_%H%M%S')
logfile="app_${ts}.log"     → app_20260609_143005.log

# ── bash built-in — no fork, fast in loops ───────────────────
```

```
printf -v ts '%(%Y%m%d_%H%M%S)T' -1
# -1 = current time. -2 = time shell was invoked. Any epoch int works.
printf -v epoch '%(%s)T' -1   # epoch seconds, no fork
```

The `printf '%(%FT%T)T'` built-in uses the same `strftime` directives as `date`, but is a bash built-in — no subprocess is spawned. Always prefer it inside loops where performance matters.

GNU date vs BSD date (macOS): The examples in this chapter use GNU `date` (Linux). macOS ships with BSD `date`, which has completely different syntax for date arithmetic and parsing. Key differences: BSD uses `-j -f FORMAT INPUT` to parse, and `-v +Nd` for arithmetic. If you write scripts that must run on both, the safest approach is to convert everything to epoch seconds immediately and do arithmetic purely in integers — that part is fully portable.

2 — Epoch Seconds: The Foundation of Date Arithmetic

Unix epoch time is the number of seconds since 1970-01-01 00:00:00 UTC. It's a plain integer — no timezone confusion, no calendar edge cases, just a number you can add, subtract, and compare with standard shell arithmetic. Every robust date calculation should start here.

Getting epoch seconds — current time and from a date string

```
# ── Current epoch ──────────────────────────────────────
date '+%s'                          → 1749470405
printf -v now '%(%s)T' -1          # no fork

# ── Parse a date string to epoch (GNU date) ────────────
date -d '2026-06-09'          '+%s'  → 1749427200
date -d '2026-06-09 14:30:00' '+%s'  → 1749479400
date -d 'yesterday'           '+%s'
date -d 'next Monday'         '+%s'
date -d '3 days ago'          '+%s'
date -d '2026-06-09 + 7 days' '+%s'

# ── Parse from a file's modification time ──────────────
date -r /etc/hosts '+%s'            # mtime as epoch (GNU)
stat -c '%Y' /etc/hosts            # same, via stat (more portable)

# ── Convert epoch back to a human date ─────────────────
epoch=1749479400
date -d "@${epoch}"                  → Tue Jun  9 14:30:00 UTC 2026
date -d "@${epoch}" '+%Y-%m-%d %H:%M:%S'   → 2026-06-09 14:30:00
# The @ prefix means "interpret as epoch seconds"
```

```bash
# ── Portable: extract individual fields from epoch ──────────
# Works on both GNU and BSD date
epoch_to_parts() {
    local epoch=$1
    IFS=' ' read -r year month day hour min sec \
        <<< "$(date -d "@$epoch" '+%Y %m %d %H %M %S')"
    printf 'year=%s month=%s day=%s hour=%s min=%s sec=%s\n' \
        "$year" "$month" "$day" "$hour" "$min" "$sec"
}
```

3 — Date Arithmetic

Once you have epoch seconds, arithmetic is trivial — it's just integer addition and subtraction. The key constants: 60 seconds/minute, 3600 seconds/hour, 86400 seconds/day. For month and year arithmetic, delegate back to `date -d` since months have variable lengths.

Adding and subtracting time periods

```bash
# ── Add/subtract fixed intervals (pure arithmetic) ──────────
now=$(date '+%s')

one_hour_ago=$(( now - 3600 ))
one_day_ago=$(( now - 86400 ))
one_week_ago=$(( now - 86400 * 7 ))
in_30_days=$(( now + 86400 * 30 ))

# Convert result back to readable date
date -d "@$one_day_ago" '+%Y-%m-%d %H:%M:%S'

# ── Month/year arithmetic — delegate to date -d ─────────────
# Month lengths vary; don't do month arithmetic with 30*86400
next_month=$(date -d 'now + 1 month' '+%Y-%m-%d')
prev_month=$(date -d 'now - 1 month' '+%Y-%m-%d')
next_year=$(date -d 'now + 1 year'  '+%Y-%m-%d')

# ── First and last day of current month ─────────────────────
first_day=$(date '+%Y-%m-01')
last_day=$(date -d "$(date '+%Y-%m-01') + 1 month - 1 day" '+%Y-%m-%d')
# Alternative: next month's first day minus 1 second
eom_epoch=$(date -d "$(date '+%Y-%m-01') + 1 month - 1 second" '+%s')
```

```
# ── Start/end of current day ──────────────────────────
today_start=$(date '+%Y-%m-%d 00:00:00')
today_start_epoch=$(date -d "$today_start" '+%s')
today_end_epoch=$(( today_start_epoch + 86400 - 1 ))
```

Calculating differences between two dates

Duration and age calculations

```
# ── Difference between two epoch values ──────────────
start=$(date -d '2026-01-01' '+%s')
end=$(date -d '2026-06-09' '+%s')
diff_seconds=$(( end - start ))
diff_days=$(( diff_seconds / 86400 ))
diff_hours=$(( diff_seconds / 3600 ))
echo "$diff_days days"              → 159 days

# ── Human-readable duration from seconds ─────────────
format_duration() {
    local total=$1
    local days=$(( total / 86400 ))
    local hours=$(( (total % 86400) / 3600 ))
    local mins=$(( (total % 3600) / 60 ))
    local secs=$(( total % 60 ))

    local result=""
    (( days  > 0 )) && result+="${days}d "
    (( hours > 0 )) && result+="${hours}h "
    (( mins  > 0 )) && result+="${mins}m "
    result+="${secs}s"
    printf '%s' "$result"
}

format_duration 90061             → 1d 1h 1m 1s
format_duration 3723              → 1h 2m 3s
format_duration 45                → 45s

# ── Script execution timer ────────────────────────────
T_START=$SECONDS      # $SECONDS counts seconds since shell started
# ... do work ...
elapsed=$(( SECONDS - T_START ))
echo "Completed in $(format_duration $elapsed)"

# Higher resolution: use date +%s%N for nanoseconds
```

```
t0=$(date '+%s%N')
# ... do work ...
t1=$(date '+%s%N')
ms=$(( (t1 - t0) / 1000000 ))
echo "Elapsed: ${ms}ms"
```

4 — Comparing Timestamps

Since epoch seconds are integers, all comparisons use standard integer operators: `-lt`, `-gt`, `-eq`, or `(( ))` arithmetic. This is far more reliable than comparing date strings lexicographically (which only works safely with ISO format `YYYY-MM-DD`).

Is a timestamp past? Is a file too old? Is it business hours?

```
# ── Is a deadline past? ──────────────────────────────
deadline=$(date -d '2026-12-31 23:59:59' '+%s')
now=$(date '+%s')

if (( now > deadline )); then
    echo "Deadline has passed"
else
    remaining=$(( deadline - now ))
    echo "$(format_duration $remaining) until deadline"
fi

# ── Is a file older than N days? ─────────────────────
file_older_than() {
    local file="$1"
    local days="$2"
    [[ -e "$file" ]] || return 0  # missing file: treat as infinitely old
    local mtime
    mtime=$(stat -c '%Y' "$file")
    local cutoff=$(( $(date '+%s') - days * 86400 ))
    (( mtime < cutoff ))
}

if file_older_than /var/cache/myapp/data.cache 7; then
    echo "Cache is stale — refreshing"
    refresh_cache
fi

# ── Find all files modified in the last N minutes ────
```

```
cutoff=$(( $(date '+%s') - 30 * 60 ))   # 30 minutes ago
while IFS= read -r -d '' f; do
    mtime=$(stat -c '%Y' "$f")
    (( mtime >= cutoff )) && echo "Recent: $f"
done < <(find /var/log -name '*.log' -print0)
# Better: let find do it
find /var/log -name '*.log' -newer <(date -d '30 minutes ago')

# ── Compare two ISO date strings directly (lexicographic — safe for YYYY-MM-DD)
d1="2026-01-15"
d2="2026-06-09"
[[ "$d1" < "$d2" ]] && echo "$d1 is earlier"
# This ONLY works reliably with YYYY-MM-DD or YYYYMMDD — never with locale dates
```

5 — Timezone Handling

The `TZ` environment variable controls which timezone `date` uses. You can set it inline for a single command, export it for the current script, or use it to convert between zones. Epoch seconds are always UTC — the timezone only affects how they're *displayed* or *parsed*.

Working with timezones

```
# ── Display current time in multiple timezones ──────────────
TZ='UTC'              date '+%H:%M %Z'   → 13:30 UTC
TZ='America/New_York' date '+%H:%M %Z'   → 09:30 EDT
TZ='Europe/London'    date '+%H:%M %Z'   → 14:30 BST
TZ='Asia/Tokyo'       date '+%H:%M %Z'   → 22:30 JST

# ── Convert a timestamp from one zone to another ────────────
# Step 1: parse the local time as epoch (forces UTC interpretation)
# Step 2: display with the target TZ
ny_time="2026-06-09 09:30:00"
epoch=$(TZ='America/New_York' date -d "$ny_time" '+%s')
tokyo_time=$(TZ='Asia/Tokyo' date -d "@$epoch" '+%Y-%m-%d %H:%M %Z')
echo "$ny_time ET  =  $tokyo_time"

# ── Ensure a script always uses UTC ─────────────────────────
export TZ='UTC'
# All subsequent date calls use UTC regardless of system timezone
# Critical for cron jobs that run on servers across multiple timezones

# ── List available timezone names ───────────────────────────
```

```
timedatectl list-timezones          # systemd systems
ls /usr/share/zoneinfo/             # all systems

# ── Check current system timezone ──────────────────────
date '+%Z %z'                       → UTC +0000
timedatectl | grep 'Time zone'      → Time zone: Europe/London (BST, +0100)

# ── DST-safe "is it past 9am in New York?" ─────────────
# DON'T use fixed offsets like UTC-5 — DST changes them
# DO use named timezones — the OS handles DST automatically
ny_hour=$(TZ='America/New_York' date '+%-H')  # %-H strips leading zero
(( ny_hour >= 9 && ny_hour < 17 )) && echo "NY business hours"
```

Timezone names come from the IANA tz database (also called the Olson database). Always use full names like `America/New_York`, never abbreviations like `EST` — abbreviations are ambiguous (three different countries use `CST` with different UTC offsets).

6 — Scheduling Logic: Day-of-Week and Time-Window Decisions

Scripts running under cron or as daemons often need to make decisions based on the current time: "only run on weekdays", "don't run during backup window", "send the digest every Monday". This is cleaner in the script itself than trying to encode all the logic in crontab syntax.

Day, week, and time-window decision patterns

```
# ── Is it a weekday? ──────────────────────────────────
dow=$(date '+%u')   # 1=Mon … 7=Sun  (ISO weekday, most useful)
if (( dow >= 1 && dow <= 5 )); then
    echo "Weekday — running job"
else
    echo "Weekend — skipping"
    exit 0
fi

# ── Is it Monday? (for weekly tasks) ──────────────────
if [[ "$(date '+%u')" == "1" ]]; then
    send_weekly_digest
fi

# ── Is it the first day of the month? ─────────────────
if [[ "$(date '+%d')" == "01" ]]; then
    generate_monthly_report
```

```
fi

# ── Is the current time inside a maintenance window? ──────────
# Maintenance: 02:00-04:00 UTC every night
in_maintenance_window() {
    local h m
    IFS=':' read -r h m <<< "$(TZ=UTC date '+%H:%M')"
    local mins=$(( 10#$h * 60 + 10#$m ))  # force base-10 (08, 09 are octal-safe)
    (( mins >= 120 && mins < 240 ))         # 02:00 = 120 mins, 04:00 = 240 mins
}

if in_maintenance_window; then
    echo "In maintenance window — aborting"
    exit 0
fi

# ── Wait until a specific time ────────────────────────────────
sleep_until() {
    local target_epoch
    target_epoch=$(date -d "$1" '+%s')
    local now_epoch
    now_epoch=$(date '+%s')
    local wait_secs=$(( target_epoch - now_epoch ))
    if (( wait_secs > 0 )); then
        echo "Sleeping ${wait_secs}s until $1"
        sleep "$wait_secs"
    fi
}

sleep_until "03:00 tomorrow"
sleep_until "next Monday 08:00"
```

Handling the midnight rollover problem

Time windows that span midnight

```
# A maintenance window 22:00-02:00 crosses midnight
# Naive hour comparison breaks: is 23:00 in range [22,2]?

in_overnight_window() {
    local start_h="$1"   # e.g. 22
    local end_h="$2"     # e.g. 2
    local cur_h
    cur_h=$(date '+%-H')  # current hour, no leading zero
```

```
    if (( start_h > end_h )); then
        # Window crosses midnight: in range if >= start OR < end
        (( cur_h >= start_h || cur_h < end_h ))
    else
        # Normal window (same day)
        (( cur_h >= start_h && cur_h < end_h ))
    fi
}

in_overnight_window 22 2  # 22:00–02:00 maintenance window
if in_overnight_window 22 2; then
    echo "Currently in maintenance window"
fi

# ── Is it the last day of the month? ──────────────────────
# Trickier: "tomorrow is the first of next month"
is_last_day_of_month() {
    local tomorrow_day
    tomorrow_day=$(date -d 'tomorrow' '+%d')
    [[ "$tomorrow_day" == "01" ]]
}
is_last_day_of_month && run_end_of_month_jobs
```

7 — Practical Patterns: Log Rotation and File Aging

Date-stamped log files and rotation

```
#!/usr/bin/env bash
# ── Pattern: one log file per day ─────────────────────────
LOG_DIR=/var/log/myapp
LOG_FILE="${LOG_DIR}/app_$(date '+%Y-%m-%d').log"

log() {
    printf -v _ts '%(%Y-%m-%d %H:%M:%S)T' -1   # no fork
    printf '[%s] %s\n' "$_ts" "$*" >> "$LOG_FILE"
}

# ── Delete logs older than N days ─────────────────────────
rotate_logs() {
    local dir="$1" keep_days="$2"
    local cutoff=$(( $(date '+%s') - keep_days * 86400 ))
```

```
    local file mtime deleted=0

    while IFS= read -r -d '' file; do
        mtime=$(stat -c '%Y' "$file")
        if (( mtime < cutoff )); then
            rm -f "$file"
            (( deleted++ ))
        fi
    done < <(find "$dir" -maxdepth 1 -type f -name '*.log' -print0)
    echo "Rotated: deleted $deleted logs older than $keep_days days"
}

rotate_logs "$LOG_DIR" 30

# ── Generate a date-stamped backup filename ──────────────
backup_filename() {
    local base="${1%.*}"       # strip extension
    local ext="${1##*.}"        # get extension
    local ts
    printf -v ts '%(%Y%m%d_%H%M%S)T' -1
    printf '%s_%s.%s' "$base" "$ts" "$ext"
}
backup_filename "config.yaml"    → config_20260609_143005.yaml
backup_filename "database.sql"   → database_20260609_143005.sql
```

8 — `sleep`, Fractional Seconds, and Polling Loops

sleep patterns and high-resolution timing

```
# ── sleep accepts fractional seconds (GNU sleep) ─────────
sleep 0.1           # 100ms
sleep 0.5           # 500ms
sleep 2.5           # 2.5 seconds
sleep "${POLL_INTERVAL:-5}"   # configurable interval

# ── Exponential backoff ──────────────────────────────────
retry_with_backoff() {
    local max_attempts="${1}"; shift
    local delay=1
    local attempt

    for (( attempt=1; attempt <= max_attempts; attempt++ )); do
```

```
        if "$@"; then
            return 0
        fi
        if (( attempt < max_attempts )); then
            printf 'Attempt %d/%d failed — retrying in %ds\n' \
                "$attempt" "$max_attempts" "$delay" >&2
            sleep "$delay"
            delay=$(( delay * 2 > 60 ? 60 : delay * 2 ))  # cap at 60s
        fi
    done
    return 1
}

retry_with_backoff 5 curl -sf https://api.example.com/health

# ── Poll until condition is met, with timeout ──────────────
wait_for() {
    local timeout="$1"; shift   # timeout in seconds
    local deadline=$(( $(date '+%s') + timeout ))

    while (( $(date '+%s') < deadline )); do
        "$@" && return 0
        sleep 2
    done
    printf 'Timed out after %ds waiting for: %s\n' "$timeout" "$*" >&2
    return 1
}

# Wait up to 60s for a service to become healthy
wait_for 60 curl -sf http://localhost:8080/health
echo "Service is up"

# ── The SECONDS built-in — no fork, precise to 1 second ────
T0=$SECONDS
# ... work ...
echo "Elapsed: $(( SECONDS - T0 ))s"

# $SECONDS can be assigned to reset the counter:
SECONDS=0
# ... work ...
echo "Elapsed: ${SECONDS}s"
```

9 — Portability: GNU vs BSD `date`

🍎 BSD date (macOS) — different syntax for arithmetic and parsing

```
# BSD date does NOT support -d "string" for parsing
# Instead use -j (don't set date) and -f FORMAT

# Parse a date string to epoch on macOS:
date -j -f '%Y-%m-%d' '2026-06-09' '+%s'    → epoch int

# Parse with time on macOS:
date -j -f '%Y-%m-%d %H:%M:%S' '2026-06-09 14:30:00' '+%s'

# Date arithmetic on macOS with -v:
# -v +7d = add 7 days, -v -1m = subtract 1 month
date -v +7d '+%Y-%m-%d'             # 7 days from now
date -v -1m '+%Y-%m-%d'             # 1 month ago
date -v +1y -v +3d '+%Y-%m-%d'     # 1 year + 3 days from now

# Convert epoch to date on macOS:
date -r 1749479400 '+%Y-%m-%d %H:%M:%S'   # -r instead of @epoch
```

🐧 Writing portable date code for both GNU and BSD

```
# Detect which date we have
if date --version >/dev/null 2>&1; then
    DATE_TYPE="gnu"
else
    DATE_TYPE="bsd"
fi

# Portable: parse a YYYY-MM-DD string to epoch
date_to_epoch() {
    if [[ "$DATE_TYPE" == "gnu" ]]; then
        date -d "$1" '+%s'
    else
        date -j -f '%Y-%m-%d' "$1" '+%s'
    fi
}

# Portable: display epoch as formatted date
epoch_to_date() {
    if [[ "$DATE_TYPE" == "gnu" ]]; then
        date -d "@$1" "${2:+%Y-%m-%d}"
    else
```

```
        date -r "$1" "${2:++%Y-%m-%d}"
    fi
}

# Fully portable: date arithmetic using only epoch + integer math
# Avoids calling date at all for simple add/subtract
now_epoch=$(date '+%s')            # works on both
week_ago=$(( now_epoch - 7*86400 ))  # pure arithmetic — fully portable
week_ago_date=$(epoch_to_date "$week_ago")
```

10 — Quick Reference

Key constants for arithmetic

PERIOD	SECONDS	EXPRESSION
1 minute	60	`60`
1 hour	3,600	`60 * 60`
1 day	86,400	`60 * 60 * 24`
1 week	604,800	`86400 * 7`
30 days	2,592,000	`86400 * 30` *(approx — use date -d for months)*
365 days	31,536,000	`86400 * 365` *(approx — use date -d for years)*

Essential one-liners

TASK	COMMAND
Current epoch (no fork)	`printf -v now '%(%s)T' -1`
Current timestamp for filenames	`printf -v ts '%(%Y%m%d_%H%M%S)T' -1`
Parse date string to epoch	`date -d 'YYYY-MM-DD HH:MM:SS' '+%s'`
Convert epoch to date	`date -d "@$epoch" '+%Y-%m-%d %H:%M:%S'`
N days ago (epoch)	`$(( $(date '+%s') - N * 86400 ))`
Day of week (1=Mon, 7=Sun)	`date '+%u'`
Script elapsed time	`$(( SECONDS - T0 ))`

TASK	COMMAND
File modification time (epoch)	`stat -c '%Y' FILE`
Force base-10 for hours/mins	`$(( 10#$h ))` — prevents octal interpretation of 08, 09
Sleep until a future time	`sleep $(( $(date -d "TARGET" '+%s') - $(date '+%s') ))`

🖍️ Exercises

These exercises build real utility functions and scripts you'll reach for regularly. Test each with edge cases: end of month, midnight rollovers, dates in the past and future.

EXERCISE 1

Write a function called `age_report()` that accepts a list of file paths and produces a formatted table showing each file's name, size, modification date (YYYY-MM-DD HH:MM:SS), age in a human-readable form (e.g. "3d 4h 12m"), and a status of FRESH (less than 1 day old), AGING (1–7 days), or STALE (more than 7 days). Sort output by age, oldest first.

Hint: use `stat -c '%Y %s %n'` to get mtime epoch, size, and name in one call. Convert age in seconds with your `format_duration()` function. Sort by epoch ascending using `sort -n` on the first field. Colour the STATUS column: green for FRESH, yellow for AGING, red for STALE.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
set -euo pipefail

format_duration() {
    local total=$1
    local d=$(( total / 86400 ))
    local h=$(( (total % 86400) / 3600 ))
    local m=$(( (total % 3600) / 60 ))
    local s=$(( total % 60 ))
    local out=""
    (( d > 0 )) && out+="${d}d "
    (( h > 0 || d > 0 )) && out+="${h}h "
    (( m > 0 || h > 0 || d > 0 )) && out+="${m}m "
    out+="${s}s"
    printf '%s' "$out"
}
```

```
age_report() {
    local now
    printf -v now '%(%s)T' -1

    # Collect: mtime|size|path — then sort by mtime ascending
    local data=()
    local f
    for f in "$@"; do
        [[ -e "$f" ]] || { echo "skip: $f not found" >&2; continue; }
        data+=( "$(stat -c '%Y|%s|%n' "$f")" )
    done

    # Header
    printf '\033[1m%-35s %-10s %-20s %-14s %s\033[0m\n' \
        "FILE" "SIZE" "MODIFIED" "AGE" "STATUS"
    printf '%.0s─' {1..95}; echo

    # Sort by mtime (field 1) ascending, oldest first
    while IFS='|' read -r mtime size path; do
        local age=$(( now - mtime ))
        local age_str; age_str=$(format_duration "$age")
        local mod_date; mod_date=$(date -d "@$mtime" '+%Y-%m-%d %H:%M:%S')
        local name="${path##*/}"

        local status colour
        if   (( age < 86400 ));       then status="FRESH"; colour="\033[32m"
        elif (( age < 86400 * 7 ));  then status="AGING"; colour="\033[33m"
        else                                 status="STALE"; colour="\033[31m"
        fi

        printf '%-35s %-10s %-20s %-14s %b%s\033[0m\n' \
            "$name" "$size" "$mod_date" "$age_str" "$colour" "$status"
    done < <(printf '%s\n' "${data[@]}" | sort -t'|' -k1,1n)
}

# Test with some files
age_report /etc/hostname /etc/hosts /etc/passwd /etc/fstab
```

EXERCISE 2

Write a script called `business_days.sh` that calculates the number of business days (Monday–Friday) between two dates passed as arguments in `YYYY-MM-DD` format, and also generates a list of all the business day dates between them. It should correctly skip weekends. Bonus: accept an optional third argument, a file of holidays in `YYYY-MM-DD` format (one per line), and exclude those too.

Hint: iterate day by day from start epoch to end epoch, adding 86400 each time. For each day, check `date -d "@$epoch" '+%u'` to get the weekday (6=Sat, 7=Sun). Load the holidays file into an associative array for O(1) lookup: `holidays["2026-01-01"]=1`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# business_days.sh  START_DATE  END_DATE  [HOLIDAYS_FILE]
set -euo pipefail

START="${1:?Usage: business_days.sh YYYY-MM-DD YYYY-MM-DD [holidays.txt]}"
END="${2:?Usage: business_days.sh YYYY-MM-DD YYYY-MM-DD [holidays.txt]}"
HOLIDAY_FILE="${3:-}"

# Load holidays into associative array
declare -A holidays=()
if [[ -n "$HOLIDAY_FILE" && -f "$HOLIDAY_FILE" ]]; then
    while IFS= read -r hday; do
        [[ -z "$hday" || "$hday" == '#'* ]] && continue
        holidays["$hday"]=1
    done < "$HOLIDAY_FILE"
    echo "Loaded ${#holidays[@]} holiday(s)"
fi

# Parse start/end to epoch (normalised to midnight)
start_epoch=$(date -d "$START 00:00:00" '+%s')
end_epoch=$(date -d "$END 00:00:00"   '+%s')

(( end_epoch >= start_epoch )) || {
    echo "Error: END must be >= START" >&2; exit 1
}

count=0
business_days=()
curr=$start_epoch
```

```
while (( curr <= end_epoch )); do
    dow=$(date -d "@$curr" '+%u')        # 1=Mon … 7=Sun
    ymd=$(date -d "@$curr" '+%Y-%m-%d')

    if (( dow <= 5 )) && [[ -z "${holidays[$ymd]+x}" ]]; then
        business_days+=( "$ymd" )
        (( count++ ))
    fi
    (( curr += 86400 ))
done

# Output
printf '\nBusiness days from %s to %s: \033[1m%d\033[0m\n\n' \
    "$START" "$END" "$count"
printf '%s\n' "${business_days[@]}"
```

EXERCISE 3

Write a script called `run_window.sh` that wraps any command and enforces a time window: it should only execute the command if the current time in a specified timezone falls within a configured start and end time. If outside the window it should print a clear message and exit 0 (so cron doesn't flag it as an error). Make the window configurable via environment variables: `WINDOW_TZ`, `WINDOW_START` (HH:MM), `WINDOW_END` (HH:MM), and optionally `WINDOW_DAYS` (comma-separated: `Mon,Tue,Wed,Thu,Fri`).

Hint: use `TZ="$WINDOW_TZ" date '+%H %M %u'` to get the current hour, minute, and weekday in the target timezone. Convert HH:MM to minutes-since-midnight for comparison, handling the midnight wrap. Parse `WINDOW_DAYS` with `IFS=,` and check the current weekday abbreviation.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# run_window.sh — only execute CMD if current time is in the configured window
# Env vars: WINDOW_TZ, WINDOW_START (HH:MM), WINDOW_END (HH:MM), WINDOW_DAYS
set -euo pipefail

WINDOW_TZ="${WINDOW_TZ:-UTC}"
WINDOW_START="${WINDOW_START:-09:00}"
WINDOW_END="${WINDOW_END:-17:00}"
WINDOW_DAYS="${WINDOW_DAYS:-}"       # empty = any day
```

```
(( $# > 0 )) || { echo "Usage: run_window.sh COMMAND [ARGS]" >&2; exit 1; }

# Get current time components in the target timezone
read -r cur_h cur_m cur_dow_n cur_dow_s \
    <<< "$(TZ="$WINDOW_TZ" date '+%-H %-M %u %a')"

# Minutes-since-midnight helper
hhmm_to_mins() {
    local h=$(( 10#${1%%:*} ))
    local m=$(( 10#${1##*:} ))
    printf '%d' $(( h * 60 + m ))
}

cur_mins=$(( 10#$cur_h * 60 + 10#$cur_m ))
start_mins=$(hhmm_to_mins "$WINDOW_START")
end_mins=$(hhmm_to_mins "$WINDOW_END")

# Check time window (handles overnight wrap)
in_window=0
if (( start_mins <= end_mins )); then
    (( cur_mins >= start_mins && cur_mins < end_mins )) && in_window=1
else
    (( cur_mins >= start_mins || cur_mins < end_mins )) && in_window=1
fi

# Check day-of-week constraint
day_ok=1
if [[ -n "$WINDOW_DAYS" ]]; then
    day_ok=0
    IFS=',' read -r -a allowed_days <<< "$WINDOW_DAYS"
    for d in "${allowed_days[@]}"; do
        [[ "${d^}" == "${cur_dow_s^}" ]] && { day_ok=1; break; }
    done
fi

# Print summary
printf '[window] TZ=%s  now=%s %02d:%02d  window=%s-%s  days=%s\n' \
    "$WINDOW_TZ" "$cur_dow_s" "$cur_h" "$cur_m" \
    "$WINDOW_START" "$WINDOW_END" "${WINDOW_DAYS:-any}" >&2

if (( in_window && day_ok )); then
    printf '[window] \033[32mIn window\033[0m — running: %s\n' "$*" >&2
```

```
    exec "$@"
else
    printf '[window] \033[33mOutside window\033[0m — skipping\n' >&2
    exit 0
fi

# Example usage (in crontab, runs every 15 min but only executes in window):
# */15 * * * * WINDOW_TZ=America/New_York WINDOW_START=09:00 WINDOW_END=17:00 \
#             WINDOW_DAYS=Mon,Tue,Wed,Thu,Fri \
#             /usr/local/bin/run_window.sh /opt/myapp/process_orders.sh
```

EXERCISE 4

Write a script called `timed_backup.sh` that backs up a directory by creating a timestamped `.tar.gz` archive, then deletes archives older than a configurable retention period (`KEEP_DAYS`, default 14). It should log each action with a timestamp, report the size of the new archive and how much space was freed by deletions, and produce a one-line summary at the end. Use `printf -v` for all timestamp generation inside the script to avoid unnecessary forks.

Hint: use `printf -v ts '%(%Y%m%d_%H%M%S)T' -1` for the archive filename. After creating the archive, use `stat -c '%s'` for its size. For deletion, loop over existing archives and compare `stat -c '%Y'` mtime against the cutoff epoch. Track freed bytes in a counter.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# timed_backup.sh  SOURCE_DIR  BACKUP_DIR
set -euo pipefail

SOURCE="${1:?Usage: timed_backup.sh SOURCE_DIR BACKUP_DIR}"
DEST="${2:?Usage: timed_backup.sh SOURCE_DIR BACKUP_DIR}"
KEEP_DAYS=${KEEP_DAYS:-14}

[[ -d "$SOURCE" ]] || { echo "Error: $SOURCE not a directory" >&2; exit 1; }
mkdir -p "$DEST"

# Logging (no fork — printf -v for timestamp)
log() {
    local _ts
    printf -v _ts '%(%Y-%m-%d %H:%M:%S)T' -1
```

```
    printf '[%s] %s\n' "$_ts" "$*"
}

# Human-readable bytes
human_bytes() {
    local b=$1
    if   (( b >= 1073741824 )); then printf '%.1f GB' "$(bc <<< "scale=1; $b/107374
    elif (( b >=    1048576 )); then printf '%.1f MB' "$(bc <<< "scale=1; $b/104857
    elif (( b >=       1024 )); then printf '%.1f KB' "$(bc <<< "scale=1; $b/1024")
    else                            printf '%d B'    "$b"
    fi
}

# ── Step 1: Create archive ──────────────────────────────────
local ts
printf -v ts '%(%Y%m%d_%H%M%S)T' -1
source_name="${SOURCE##*/}"
archive="${DEST}/${source_name}_${ts}.tar.gz"

log "Starting backup: $SOURCE -> $archive"
tar -czf "$archive" -C "$(dirname "$SOURCE")" "$source_name"
archive_size=$(stat -c '%s' "$archive")
log "Archive created: $(human_bytes $archive_size)"

# ── Step 2: Delete old archives ─────────────────────────────
cutoff=$(( $(date '+%s') - KEEP_DAYS * 86400 ))
freed=0
deleted=0

while IFS= read -r -d '' old; do
    [[ "$old" == "$archive" ]] && continue  # don't delete what we just made
    mtime=$(stat -c '%Y' "$old")
    if (( mtime < cutoff )); then
        fsize=$(stat -c '%s' "$old")
        rm -f "$old"
        (( freed += fsize ))
        (( deleted++ ))
        log "Deleted old archive: ${old##*/} ($(human_bytes $fsize))"
    fi
done < <(find "$DEST" -maxdepth 1 -name "${source_name}_*.tar.gz" -print0)
```

```
# ── Summary ──────────────────────────────────────────────────
log "Done. Created: $(human_bytes $archive_size)  Freed: $(human_bytes $freed)  Del
```

📊 Intermediate Topic 6 — Structured Data: CSV, TSV, and INI

Shell scripts constantly consume and produce structured text: exports from databases, configuration files, reports. The tools are right there — `awk`, `IFS` splitting, pattern matching — but the edge cases bite hard. A CSV file with quoted fields containing commas will silently corrupt your data if you split naively. An INI file with inline comments or trailing spaces will hand you the wrong value. This chapter covers the full picture: how the formats actually work, where the traps are, and the robust patterns used in production scripts.

1 — CSV Fundamentals: Why It's Harder Than It Looks

CSV sounds simple — values separated by commas. The problem is that real CSV (RFC 4180) allows fields to contain commas, newlines, and double-quote characters, all of which break naive parsing. A field that contains a comma must be wrapped in double-quotes. A double-quote inside a quoted field must be escaped as two consecutive double-quotes ("").

❌ NAIVE — BREAKS ON QUOTED FIELDS

```
# Input: name,city,note
# Alice,"New York",no comma
# Bob,"Paris, France",has comma

while IFS=',' read -r name city note; do
    echo "city=$city"
done < data.csv
# Bob's city = '"Paris'   ← WRONG
# note       = ' France"' ← WRONG
```

✓ REAL-WORLD CSV HAS QUOTING RULES

```
# RFC 4180 rules:
# 1. Fields MAY be quoted with "
# 2. A quoted field may contain , \n "
# 3. A " inside quotes = ""
# 4. Leading/trailing spaces inside
#    quotes are part of the value
#
# "Alice","New York","says ""hi"""
# → Alice | New York | says "hi"
#
# Need a state-machine parser.
# awk's FPAT is the cleanest option.
```

The IFS comma split is only safe when you control the data. If your CSV comes from a spreadsheet export, a database dump, or any system that follows RFC 4180, you must handle quoted fields. Use the `awk FPAT` approach or pipe through a purpose-built tool like `csvkit` (`csvcut`, `csvstat`). For data you generate yourself — logs, reports — you can choose to avoid values that need quoting, or use TSV instead.

2 — Robust CSV Parsing with `awk` and `FPAT`

GNU awk (`gawk`) provides a special variable `FPAT` — Field PATtern — which instead of defining the *separator* defines what a *field looks like*. Setting it to a pattern that matches either a quoted field or an unquoted field gives you a proper CSV parser in one line.

FPAT-based CSV parsing in gawk

```
# FPAT pattern breakdown:
#   ([^,]*)           — any unquoted field (zero or more non-comma chars)
#   |                 — OR
#   ("([^"]|"")*")    — a double-quoted field (handles "" escape inside)
FPAT_CSV='([^,]*)|("([^"]|"")*")'

# ── Print every field on a separate line ──────────────────────
gawk -v FPAT='([^,]*)|("([^"]|"")*")' '
{ for (i=1; i<=NF; i++) print i, $i }
' data.csv

# ── Strip surrounding quotes from a field value ───────────────
# Fields from FPAT still include the outer quotes — strip them:
csv_unquote() {
    # Strips surrounding quotes and unescapes "" → "
    awk 'BEGIN {
        s = ARGV[1]
        if (substr(s,1,1) == "\"") s = substr(s, 2, length(s)-2)
        gsub(/""/, "\"", s)
        print s
    }' "$1"
}

# ── Practical: process a CSV with header row ──────────────────
# File: employees.csv
# id,name,department,salary
# 1,Alice,"Engineering",95000
# 2,Bob,"Sales, East",72000
# 3,"O'Brien, Carol","HR",68000

gawk -v FPAT='([^,]*)|("([^"]|"")*")' '
NR == 1 { next }            # skip header
{
    id   = $1
    name = $2; gsub(/^"|"$/, "", name)   # strip quotes
    dept = $3; gsub(/^"|"$/, "", dept)
```

```
    sal  = $4

    printf "%-4s %-20s %-20s %s\n", id, name, dept, sal
}
' employees.csv

# ── Filter: only Engineering employees ──────────────────────
gawk -v FPAT='([^,]*)|("([^"]|"")*")' '
NR == 1 { print; next }
{
    dept = $3; gsub(/^"|"$/, "", dept)
    if (dept == "Engineering") print
}
' employees.csv

# ── Extract a single column by header name ──────────────────
gawk -v FPAT='([^,]*)|("([^"]|"")*")' \
     -v col="salary" '
NR == 1 {
    for (i=1; i<=NF; i++) {
        h = $i; gsub(/^"|"$/, "", h)
        if (h == col) colidx = i
    }
    next
}
colidx { v = $colidx; gsub(/^"|"$/, "", v); print v }
' employees.csv
```

FPAT is a GNU awk extension — not available in POSIX awk or mawk. Check with gawk --version. On systems where only awk is available, fall back to a Python one-liner or csvkit.

Building a reusable csv_parse() library function

A shell function that turns a CSV row into a bash array

```
# csv_split ROW → prints fields one per line, with quotes stripped
# Pipe its output to mapfile to get an array
csv_split() {
    gawk 'BEGIN {
        FPAT = "([^,]*)|(\\"([^\\"]|\\"\\")* \\")"
        $0 = ARGV[1]
        for (i=1; i<=NF; i++) {
            v = $i
            if (substr(v,1,1) == "\"") v = substr(v, 2, length(v)-2)
            gsub(/""/, "\"", v)
```

```
            print v
        }
    }' "$1"
}

# Usage: turn a CSV row into a bash array
row='42,"Smith, John","New York","says ""hello"""'
mapfile -t fields < <(csv_split "$row")

echo "ID:   ${fields[0]}"   → 42
echo "Name: ${fields[1]}"   → Smith, John
echo "City: ${fields[2]}"   → New York
echo "Note: ${fields[3]}"   → says "hello"

# Process a whole file row by row
process_csv() {
    local file="$1"
    local header_done=0
    local row

    while IFS= read -r row; do
        (( header_done == 0 )) && { header_done=1; continue; }
        mapfile -t f < <(csv_split "$row")
        # Now f[0], f[1], f[2]... hold the clean field values
        printf 'id=%s name=%s\n' "${f[0]}" "${f[1]}"
    done < "$file"
}
```

3 — Generating CSV Output

Writing CSV is easier than reading it — you control the quoting. The safest rule: always quote fields that may contain commas, double-quotes, or newlines; double up any literal double-quotes inside them. For simple data you know is clean, quoting only when necessary is fine.

Writing safe CSV from shell scripts

```
# ── csv_field: quote a single value for CSV output ──────────
csv_field() {
    local val="$1"
    # Always quote — simplest and safest
    val="${val//\"/\"\"}"   # escape " → ""
    printf '"%s"' "$val"
```

```
}

# ── csv_row: turn args into a CSV line ──────────────────────
csv_row() {
    local first=1
    for arg in "$@"; do
        (( first )) || printf ','
        csv_field "$arg"
        first=0
    done
    printf '\n'
}

# ── Generate a CSV report ────────────────────────────────────
{
    csv_row "hostname" "cpu_pct" "mem_pct" "disk_pct" "timestamp"

    while IFS= read -r host; do
        # Collect metrics for $host (simplified)
        cpu=$(ssh "$host" "top -bn1 | awk '/Cpu/ {print 100-\$8}'" 2/dev/null)
        ts=$(date '+%Y-%m-%d %H:%M:%S')
        csv_row "$host" "${cpu:-N/A}" "N/A" "N/A" "$ts"
    done < hosts.txt
} > report.csv

# ── Append to an existing CSV safely ─────────────────────────
csv_append() {
    local file="$1"; shift
    if [[ ! -f "$file" ]]; then
        # Create with header on first call if header provided as first arg
        touch "$file"
    fi
    csv_row "$@" >> "$file"
}

# ── Column extraction without gawk (safe only for simple CSV) ─
# cut -d, -f2 — works only when no quoted commas exist
cut -d',' -f2 employees.csv

# With gawk FPAT — handles quoted commas
gawk -v FPAT='([^,]*)|("([^"]|"")*")' '{v=$2; gsub(/^"|"$/,"",v); print v}' employees.csv
```

4 — TSV: Tab-Separated Values

TSV uses a tab character as the delimiter. Tabs almost never appear in real data, which means quoting rules are usually unnecessary and parsing with `IFS=$'\t'` is safe. Many tools (`sort`, `join`, `awk`) integrate cleanly with tab-delimited data. When you control the format, TSV is often the better choice over CSV for shell processing.

TSV reading, writing, and conversion

```
# ── Reading TSV ──────────────────────────────────────────
while IFS=$'\t' read -r id name dept salary; do
    printf '%-4s %-20s %-15s %s\n' "$id" "$name" "$dept" "$salary"
done < employees.tsv

# ── Writing TSV ──────────────────────────────────────────
tsv_row() {
    local IFS=$'\t'
    printf '%s\n' "$*"
}
tsv_row "id" "name" "department" "salary"
tsv_row "1"  "Alice" "Engineering" "95000"

# ── Converting CSV to TSV ────────────────────────────────
# Simple (only safe when no embedded commas/newlines in fields)
tr ',' $'\t' < simple.csv > simple.tsv

# Robust (uses gawk FPAT to parse, then re-joins with tab)
gawk -v FPAT='([^,]*)|("([^"]|"")*")' '
{
    for (i=1; i<=NF; i++) {
        v = $i
        gsub(/^"|"$/, "", v)   # strip surrounding quotes
        gsub(/""/, "\"", v)    # unescape doubled quotes
        printf "%s%s", (i>1?"\t":""), v
    }
    print ""
}
' input.csv > output.tsv

# ── Sort TSV by column ───────────────────────────────────
sort -t$'\t' -k3,3       employees.tsv   # sort by col 3 (alphabetic)
sort -t$'\t' -k4,4n -r   employees.tsv   # sort by col 4 (numeric, desc)

# ── join two TSV files on a common key ──────────────────
# employees.tsv:  id  name  dept_id
# departments.tsv: dept_id  dept_name  budget
# Both must be sorted on the join field first
```

```
join -t$'\t' -1 3 -2 1 \
    <(sort -t$'\t' -k3,3 employees.tsv) \
    <(sort -t$'\t' -k1,1 departments.tsv)
```

5 — INI and Config File Parsing

INI files are ubiquitous: `/etc/mysql/my.cnf`, `~/.gitconfig`, `php.ini`, `smb.conf`, countless application configs. The format has no strict standard but the conventions are consistent: `[section]` headers, `key=value` or `key = value` pairs, and `#` or `;` line comments. The traps are whitespace around the `=`, inline comments, multi-line values, and section awareness.

Reading INI files with awk

```
# Sample config.ini:
# [database]
# host = db.example.com
# port = 5432
# name = myapp_prod   # production DB
#
# [cache]
# host = redis.example.com
# port = 6379

# ── Get a single value: ini_get FILE SECTION KEY ──────────────
ini_get() {
    local file="$1" section="$2" key="$3"
    awk -v s="$section" -v k="$key" '
    /^\[/ {
        gsub(/[\[\]]/, "")
        cur = $0
        next
    }
    cur == s && /=/ {
        # Split on first = only
        eq = index($0, "=")
        key_part = substr($0, 1, eq-1)
        val_part = substr($0, eq+1)
        # Trim whitespace
        gsub(/^[[:space:]]+|[[:space:]]+$/, "", key_part)
        gsub(/^[[:space:]]+/, "", val_part)
        # Strip inline comment (# or ;)
        sub(/[[:space:]]+[#;].*$/, "", val_part)
```

```
        # Trim trailing whitespace
        gsub(/[[:space:]]+$/, "", val_part)
        if (key_part == k) { print val_part; exit }
    }
    ' "$file"
}

db_host=$(ini_get config.ini database host)
db_port=$(ini_get config.ini database port)
db_name=$(ini_get config.ini database name)
echo "Connecting to $db_host:$db_port/$db_name"

# ── Load an entire section into bash variables ──────────────
ini_load_section() {
    local file="$1" section="$2" prefix="${3:-}"
    local k v
    while IFS='=' read -r k v; do
        # Strip whitespace and inline comments
        k="${k// /}"
        v="${v#"${v%%[![:space:]]*}"}"   # ltrim
        v="${v%%[[:space:]]*#*}"          # strip inline comment
        v="${v%"${v##*[![:space:]]}"}"   # rtrim
        [[ -z "$k" ]] && continue
        declare -g "${prefix}${k}=${v}"
    done < <(awk -v s="$section" '
        /^\[/ { gsub(/[\[\]]/,""); cur=$0; next }
        cur==s && /=/ { print }
        cur==s && /^\s*($|[#;])/ { next }
    ' "$file")
}

# Loads database.host, database.port, etc. as shell vars with prefix
ini_load_section config.ini database db_
echo "host=$db_host  port=$db_port"
```

Writing and updating INI values

Updating a key in an INI file in-place

```
# ini_set FILE SECTION KEY VALUE
# Updates the key if it exists; inserts it if it doesn't
ini_set() {
    local file="$1" section="$2" key="$3" value="$4"
```

```
    if grep -q "^\[${section}\]" "$file"; then
        # Section exists — try to update existing key
        if awk -v s="$section" -v k="$key" '
            /^\[/ { cur=substr($0,2,length($0)-2) }
            cur==s && $0 ~ "^[[:space:]]*"k"[[:space:]]*=" { found=1 }
            END { exit !found }
        ' "$file"; then
            # Key exists — update it with sed
            sed -i "/^\[${section}\]/,/^\[/" \
                -e "/^\[${section}\]/,/^\[/ s|^\([[:space:]]*${key}[[:space:]]*=\).*|\1 ${va
                "$file"
        else
            # Key missing — append to section
            sed -i "/^\[${section}\]/a ${key} = ${value}" "$file"
        fi
    else
        # Section missing — append at end of file
        printf '\n[%s]\n%s = %s\n' "$section" "$key" "$value" >> "$file"
    fi
}

ini_set config.ini database port 5433
ini_set config.ini database timeout 30    # new key
ini_set config.ini newapp debug true       # new section + key

# ── List all sections in a file ──────────────────────
ini_sections() {
    grep -oP '(?<=^\[)[^\]]*' "$1"
}

# ── List all keys in a section ───────────────────────
ini_keys() {
    awk -v s="$2" '
        /^\[/ { gsub(/[\[\]]/,""); cur=$0; next }
        cur==s && /=/ {
            eq=index($0,"="); k=substr($0,1,eq-1)
            gsub(/^[[:space:]]+|[[:space:]]+$/,"",k)
            print k
        }
    ' "$1"
}

ini_sections config.ini          → database
                                   cache
ini_keys     config.ini database  → host
```

```
port
name
```

6 — Key=Value and .env Files

The simpler cousin of INI: a flat file of `KEY=VALUE` pairs with no sections. Docker Compose, systemd unit `EnvironmentFile=`, and countless deployment tools use this format. Bash can `source` it directly — but only after sanitising it, because a malicious or malformed `.env` file can execute arbitrary code on `source`.

Safely loading .env files

```
# ── UNSAFE: direct source runs any code in the file ──────────
# source .env        ← NEVER do this with untrusted files
# . .env             ← same risk

# ── Safe: parse and export only valid KEY=VALUE lines ────────
load_env() {
    local envfile="${1:-.env}"
    [[ -f "$envfile" ]] || { echo "$envfile not found" >&2; return 1; }

    local key val line
    while IFS= read -r line; do
        # Skip blanks and comments
        [[ "$line" =~ ^[[:space:]]*($|#) ]] && continue
        # Must match KEY=VALUE with valid identifier key
        [[ "$line" =~ ^([A-Za-z_][A-Za-z0-9_]*)=(.*)$ ]] || continue
        key="${BASH_REMATCH[1]}"
        val="${BASH_REMATCH[2]}"
        # Strip optional surrounding quotes
        [[ "$val" =~ ^\"(.*)\"$ ]] && val="${BASH_REMATCH[1]}"
        [[ "$val" =~ ^\'(.*)\'$ ]] && val="${BASH_REMATCH[1]}"
        # Strip inline comment (only outside quotes)
        val="${val%%[[:space:]]*\#*}"
        export "${key}=${val}"
    done < "$envfile"
}

load_env .env.production
echo "DB_HOST=$DB_HOST"

# ── source is fine when you wrote the file yourself ──────────
# In CI or deploy scripts, sourcing a file you generated is OK
```

```
# Just never source files from user input or external sources

# ── Generate a .env file ──────────────────────────────────
write_env() {
    local file="$1"; shift
    # Args: KEY VALUE KEY VALUE ...
    {
        printf '# Generated %s\n' "$(date -Is)"
        while (( $# >= 2 )); do
            k="$1" v="$2"; shift 2
            # Quote if value contains spaces, #, or special chars
            [[ "$v" =~ [[:space:]#\"] ]] && v='"'"${v//\"/\\\"}"'"'
            printf '%s=%s\n' "$k" "$v"
        done
    } > "$file"
    chmod 600 "$file"   # env files often contain secrets
}
write_env .env.staging \
    DB_HOST staging-db.internal \
    DB_PORT 5432 \
    APP_SECRET "my secret value" \
    DEBUG false
```

7 — Generating Structured Reports

A common pattern in ops scripts: collect data from multiple sources, assemble it into a structured table, and emit it as either a human-readable aligned table or a machine-readable CSV. The key is separating data collection from formatting.

Aligned console table + CSV output from the same data

```
#!/usr/bin/env bash
# disk_report.sh — show disk usage for a list of paths
set -euo pipefail

FORMAT="${FORMAT:-table}"   # table | csv | tsv
PATHS=( "${@:-/}" )

# ── Column definitions: name, width, awk-expression ──────
declare -A COL_W=( [mount]=20 [total]=8 [used]=8 [avail]=8 [pct]=6 )

# ── Collect data from df ──────────────────────────────────
```

```
# df -h outputs: Filesystem Size Used Avail Use% Mounted-on
collect_data() {
    df -h "${PATHS[@]}" 2>/dev/null | tail -n +2 | \
    awk '{print $6, $2, $3, $4, $5}'  # mount total used avail pct
}

# ── Render as aligned table ──────────────────────────────
render_table() {
    local w_m=${COL_W[mount]} w_t=${COL_W[total]}
    local w_u=${COL_W[used]}  w_a=${COL_W[avail]} w_p=${COL_W[pct]}
    local sep; sep=$(printf '%*s' $(( w_m + w_t + w_u + w_a + w_p + 5 )) '' | tr ' ' '─')

    printf "\033[1m%-${w_m}s %${w_t}s %${w_u}s %${w_a}s %${w_p}s\033[0m\n" \
        MOUNT TOTAL USED AVAIL PCT
    printf '%s\n' "$sep"

    while read -r mount total used avail pct; do
        # Colour-code by usage percentage
        pct_n=${pct//%/}
        local colour='\033[0m'
        (( pct_n >= 90 )) && colour='\033[31m'
        (( pct_n >= 75 && pct_n < 90 )) && colour='\033[33m'

        printf "%-${w_m}s %${w_t}s %${w_u}s %${w_a}s ${colour}%${w_p}s\033[0m\n" \
            "$mount" "$total" "$used" "$avail" "$pct"
    done < <(collect_data)
}

# ── Render as CSV ────────────────────────────────────────
render_csv() {
    csv_row mount total used avail pct
    while read -r mount total used avail pct; do
        csv_row "$mount" "$total" "$used" "$avail" "$pct"
    done < <(collect_data)
}

# ── Dispatch ─────────────────────────────────────────────
case "$FORMAT" in
    table) render_table ;;
    csv)   render_csv   ;;
    tsv)   render_csv | tr ',' $'\t' ;;  # quick for clean data
    *)     echo "Unknown FORMAT: $FORMAT" >&2; exit 1 ;;
esac

# Usage:
```

```
# ./disk_report.sh /var /home /tmp
# FORMAT=csv ./disk_report.sh / /var
```

8 — Quick Reference

TASK	TOOL / PATTERN	NOTES
Parse CSV with quoted fields	`gawk FPAT='([^,]*)|("([^"]|"")*")'`	GNU awk only
Simple CSV split (no quotes)	`IFS=',' read -r f1 f2 f3`	Only safe when no quoted commas
Extract CSV column by name	`gawk -v FPAT=... -v col="name" 'NR==1{...} {print $colidx}'`	
Sort TSV by column N	`sort -t$'\t' -kN,N`	Add `n` flag for numeric sort
Join two TSV files	`join -t$'\t' -1 N -2 M file1 file2`	Files must be pre-sorted
Read INI value	`ini_get FILE SECTION KEY`	awk-based function
Update INI value	`ini_set FILE SECTION KEY VALUE`	Uses sed in-place
Load .env safely	Regex-validate keys, then `export "$key=$val"`	Never `source` untrusted files
Quote a CSV field	`val="${val//\"/\"\"}"; printf '"%s"' "$val"`	Always safe to over-quote
CSV → TSV (no embedded commas)	`tr ',' $'\t'`	
CSV → TSV (with quoted commas)	`gawk FPAT=... + gsub + printf with \t`	

> **When to use a real tool instead:** *For anything beyond simple filtering and extraction, reach for `csvkit` (`pip install csvkit`) — it provides `csvcut`, `csvjoin`, `csvstat`, `csvsql`, and handles encoding, BOM markers, and edge cases you haven't thought of yet. For Python users, `pandas` and its `read_csv()` are the standard. Shell CSV parsing is the right tool for quick one-offs and environments where you can't install extra packages.*

🖍️ Exercises

Each exercise builds a utility you'd actually reach for. Test with edge cases: values containing commas, empty fields, sections with missing keys, files with Windows-style CRLF line endings.

EXERCISE 1

Write a function called `csv_summary()` that accepts a CSV file path and prints a summary of its structure: number of rows (excluding header), number of columns, each column name with its index, and for numeric columns the min, max, and average. Use `gawk` with `FPAT` so it handles quoted fields correctly.

Hint: in the NR==1 block, capture column names into an array. For data rows, try to detect numeric columns by checking if the value matches `/^-?[0-9.]+$/`. Track running sum, min, and max per column in arrays. Print the summary in the END block.

▶ Show Sample Solution

SAMPLE SOLUTION

```
csv_summary() {
    local file="${1:?csv_summary: file required}"
    [[ -f "$file" ]] || { echo "Not found: $file" >&2; return 1; }

    gawk '
    BEGIN { FPAT = "([^,]*)(\"([^\"]|\"\")*\")" }
    function strip(s) {
        gsub(/^[[:space:]"]+|[[:space:]"]+$/, "", s)
        return s
    }
    NR == 1 {
        ncols = NF
        for (i=1; i<=NF; i++) hdr[i] = strip($i)
        next
    }
    {
        rows++
        for (i=1; i<=NF; i++) {
            v = strip($i)
            if (v ~ /^-?[0-9]+(\.[0-9]+)?$/) {
                is_num[i] = 1
                sum[i] += v
                if (!(i in mn) || v+0 < mn[i]) mn[i] = v+0
                if (!(i in mx) || v+0 > mx[i]) mx[i] = v+0
            }
        }
```

```
    }
    END {
        print "File:    " FILENAME
        print "Rows:    " rows
        print "Columns: " ncols
        print ""
        printf "%-4s %-20s %-10s %s\n", "IDX", "COLUMN", "TYPE", "STATS"
        printf "%s\n", "————————————————————————————————————————————————————"
        for (i=1; i<=ncols; i++) {
            if (is_num[i]) {
                avg = (rows > 0) ? sum[i] / rows : 0
                printf "%-4d %-20s %-10s min=%-8g max=%-8g avg=%.2f\n",
                    i, hdr[i], "numeric", mn[i], mx[i], avg
            } else {
                printf "%-4d %-20s %-10s\n", i, hdr[i], "text"
            }
        }
    }
    ' "$file"
}

# Test:
printf 'id,name,score,city\n1,Alice,92,"New York"\n2,Bob,78,"Paris"\n3,Carol,85,Lon
csv_summary /tmp/test.csv
```

EXERCISE 2

Write a script called `csv_filter.sh` that filters rows from a CSV file based on a column name and a value. Usage: `csv_filter.sh FILE COLUMN VALUE`. It should print the header row, then all rows where the named column exactly matches VALUE (case-insensitive). Handle quoted fields correctly and strip surrounding quotes when comparing. Support an optional `--contains` flag that does a substring match instead of an exact match.

Hint: in NR==1 find the column index by name. In data rows, strip quotes from the target field before comparing. Store the `--contains` flag in a variable passed via `-v`. Use `tolower()` in awk for case-insensitive comparison.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# csv_filter.sh [--contains] FILE COLUMN VALUE
set -euo pipefail

MATCH_TYPE="exact"
if [[ "${1:-}" == "--contains" ]]; then
    MATCH_TYPE="contains"
    shift
fi

FILE="${1:?Usage: csv_filter.sh [--contains] FILE COLUMN VALUE}"
COL="${2:?column name required}"
VAL="${3:?value required}"

[[ -f "$FILE" ]] || { echo "Error: $FILE not found" >&2; exit 1; }

gawk -v col="$COL" -v val="$VAL" -v mtype="$MATCH_TYPE" '
BEGIN {
    FPAT = "([^,]*)(\"([^\"]|\"\")*\")"
    colidx = 0
    val_l = tolower(val)
}

function strip(s) {
    gsub(/^[[:space:]"]+|[[:space:]"]+$/, "", s)
    gsub(/""/, "\"", s)
    return s
}

NR == 1 {
    for (i=1; i<=NF; i++) {
        h = strip($i)
        if (tolower(h) == tolower(col)) { colidx = i }
    }
    if (!colidx) {
        print "Error: column \"" col "\" not found" > "/dev/stderr"
        exit 1
    }
    print   # always print header
    next
}
```

```
{
    if (!colidx) next
    field = strip($colidx)
    field_l = tolower(field)

    if (mtype == "exact" && field_l == val_l) print
    else if (mtype == "contains" && index(field_l, val_l)) print
}
' "$FILE"
```

EXERCISE 3

Write a complete config library in a file called `lib_config.sh` designed to be sourced by other scripts. It should provide: `config_load FILE` (loads all sections into `CONF[section.key]` associative array), `config_get SECTION KEY [DEFAULT]` (retrieves a value, returns DEFAULT if missing), `config_require SECTION KEY` (dies with an error if the key is absent), and `config_dump` (prints all loaded values sorted). Include a self-test that runs when the file is executed directly.

Hint: use `declare -gA CONF` to create a global associative array. The key format is `"${section}.${key}"`. In `config_get`, use `"${CONF[$section.$key]:-$default}"`. The `${BASH_SOURCE[0]} == $0` trick from Topic 4 triggers the self-test.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# lib_config.sh — INI config library
# Source this file; if executed directly it runs self-tests

declare -gA CONF=()
_CONF_LOADED=""

# config_load FILE — parse INI into CONF[section.key]
config_load() {
    local file="${1:?config_load: file required}"
    [[ -f "$file" ]] || { printf 'config_load: %s not found\n' "$file" >&2; return
    CONF=()   # reset
    _CONF_LOADED="$file"

    local cur_section=""
    local line k v
```

```
    while IFS= read -r line; do
        # Remove CR (handle Windows CRLF)
        line="${line//$'\r'/}"
        # Skip blank lines and comments
        [[ "$line" =~ ^[[:space:]]*($|[#;]) ]] && continue
        # Section header
        if [[ "$line" =~ ^\[([A-Za-z0-9_.-]+)\] ]]; then
            cur_section="${BASH_REMATCH[1]}"
            continue
        fi
        # key = value (only inside a section)
        [[ -z "$cur_section" ]] && continue
        [[ "$line" =~ ^[[:space:]]*([A-Za-z0-9_./-]+)[[:space:]]*=[[:space:]]*(.*)
        k="${BASH_REMATCH[1]}"
        v="${BASH_REMATCH[2]}"
        # Strip inline comment
        v="${v%%[[:space:]][#;]*}"
        # Strip trailing whitespace
        v="${v%"${v##*[![:space:]]}"}"
        # Strip surrounding quotes
        [[ "$v" =~ ^\"(.*)\"$ ]] && v="${BASH_REMATCH[1]}"
        [[ "$v" =~ ^\'(.*)\'$ ]] && v="${BASH_REMATCH[1]}"
        CONF["${cur_section}.${k}"]="$v"
    done < "$file"
}

# config_get SECTION KEY [DEFAULT]
config_get() {
    local sec="$1" key="$2" default="${3:-}"
    local ckey="${sec}.${key}"
    if [[ -v "CONF[$ckey]" ]]; then
        printf '%s' "${CONF[$ckey]}"
    else
        printf '%s' "$default"
    fi
}

# config_require SECTION KEY — exits with error if missing
config_require() {
    local sec="$1" key="$2"
    [[ -v "CONF[${sec}.${key}]" ]] || {
        printf 'config_require: [%s] %s is required but not set\n' "$sec" "$key" >&
```

```
        return 1
    }
    printf '%s' "${CONF[${sec}.${key}]}"
}

# config_dump — print all loaded values
config_dump() {
    printf '# Config loaded from: %s\n' "$_CONF_LOADED"
    local k
    for k in $(printf '%s\n' "${!CONF[@]}" | sort); do
        printf '%-40s = %s\n' "$k" "${CONF[$k]}"
    done
}

# ── Self-test (runs only when executed, not sourced) ──────────
if [[ "${BASH_SOURCE[0]}" == "$0" ]]; then
    tmpf=$(mktemp /tmp/lib_config_test.XXXXXX.ini)
    trap "rm -f '$tmpf'" EXIT

    cat > "$tmpf" <<'EOF'
[database]
host = db.example.com  # primary DB
port = 5432
name = "myapp prod"

[cache]
host = redis.example.com
port = 6379
EOF

    pass=0; fail=0
    assert_eq() {
        if [[ "$1" == "$2" ]]; then
            printf '\033[32m✓\033[0m %s\n' "$3"; (( pass++ ))
        else
            printf '\033[31m✗\033[0m %s (got "%s", want "%s")\n' "$3" "$1" "$2"; ((
        fi
    }

    config_load "$tmpf"

    assert_eq "$(config_get database host)"    "db.example.com"  "host without inli
    assert_eq "$(config_get database port)"    "5432"            "port as string"
```

```
    assert_eq "$(config_get database name)"    "myapp prod"      "quoted value"
    assert_eq "$(config_get cache host)"       "redis.example.com" "cache host"
    assert_eq "$(config_get database missing default_val)" \
              "default_val"  "missing key returns default"
    assert_eq "$(config_get nosec nokey fallback)" \
              "fallback"     "missing section returns default"

    echo; config_dump
    printf '\nResults: %d passed, %d failed\n' "$pass" "$fail"
    (( fail == 0 ))
fi
```

EXERCISE 4

Write a script called `csv_join.sh` that performs a left join on two CSV files, matching rows by a common key column. Usage: `csv_join.sh FILE1 COL1 FILE2 COL2`. The output should have all columns from FILE1, followed by all non-key columns from FILE2. Rows in FILE1 with no match in FILE2 should still appear, with empty strings for the FILE2 columns. Handle quoted fields throughout.

Hint: the cleanest approach is a two-pass awk: first pass reads FILE2 into an associative array keyed by the join field; second pass processes FILE1 and looks up each row's key. Pass both FPAT and the column indices via `-v`. To find column indices, process NR==1 of each file separately using `BEGINFILE` or by running two awk programs.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# csv_join.sh FILE1 COL1 FILE2 COL2
# Left join: all rows from FILE1, matched with FILE2 on key columns
set -euo pipefail

F1="${1:?Usage: csv_join.sh FILE1 COL1 FILE2 COL2}"
C1="${2:?col1 required}"
F2="${3:?file2 required}"
C2="${4:?col2 required}"

for f in "$F1" "$F2"; do
    [[ -f "$f" ]] || { printf 'Not found: %s\n' "$f" >&2; exit 1; }
done
```

```
gawk \
    -v f1="$F1" -v c1="$C1" \
    -v f2="$F2" -v c2="$C2" \
    '
BEGIN {
    FPAT = "([^,]*)(\"([^\"]|\"\")*\")"
    FS  = ","
}

function strip(s,    v) {
    v = s
    gsub(/^[[:space:]"]+|[[:space:]"]+$/, "", v)
    gsub(/""/, "\"", v)
    return v
}

function csv_field(s,    v) {
    # Always quote output for safety
    v = s; gsub(/"/, "\"\"", v)
    return "\"" v "\""
}

# ── Pass 1: load FILE2 into memory ──────────────────────
BEGINFILE {
    if (FILENAME == f2) in_f2 = 1
    else                in_f2 = 0
}

in_f2 && FNR == 1 {
    for (i=1; i<=NF; i++) {
        h = strip($i)
        if (tolower(h) == tolower(c2)) { f2_key_idx = i }
        f2_hdr[i] = h
    }
    f2_ncols = NF
    next
}

in_f2 {
    key = strip($f2_key_idx)
    row = ""
    for (i=1; i<=NF; i++) {
```

```
        if (i == f2_key_idx) continue   # exclude key col from f2 output
        row = row (row ? SUBSEP : "") strip($i)
    }
    f2_data[key] = row
    next
}

# ── Pass 2: process FILE1, join with FILE2 data ──────────
!in_f2 && FNR == 1 {
    for (i=1; i<=NF; i++) {
        h = strip($i)
        if (tolower(h) == tolower(c1)) f1_key_idx = i
        f1_hdr[i] = h
    }
    f1_ncols = NF
    # Print merged header
    out = ""
    for (i=1; i<=f1_ncols; i++) out = out (out?",":"") csv_field(f1_hdr[i])
    for (i=1; i<=f2_ncols; i++) {
        if (i == f2_key_idx) continue
        out = out "," csv_field(f2_hdr[i])
    }
    print out
    next
}

!in_f2 {
    key = strip($f1_key_idx)
    out = ""
    for (i=1; i<=f1_ncols; i++) out = out (out?",":"") csv_field(strip($i))

    if (key in f2_data) {
        n = split(f2_data[key], parts, SUBSEP)
        for (i=1; i<=n; i++) out = out "," csv_field(parts[i])
    } else {
        # No match: fill with empty fields
        empties = f2_ncols - 1   # -1 for the key col we excluded
        for (i=1; i<=empties; i++) out = out ","
    }
    print out
}
' "$F2" "$F1"   # F2 first so it is loaded before F1 is processed
```

```
# Example:
# employees.csv: id,name,dept_id
# departments.csv: dept_id,dept_name,budget
# ./csv_join.sh employees.csv dept_id departments.csv dept_id
```

🔧 Intermediate Topic 7 — Working with JSON and jq

JSON is the dominant data interchange format for APIs, config files, and tool output. Nearly every CLI tool now supports `--output json`, and virtually every REST API speaks it. `jq` is the shell's native JSON processor — a full filter language that can extract, transform, reshape, and build JSON without leaving the terminal. This chapter covers the complete jq filter syntax, shell variable injection, building JSON from shell data, and the full pattern of calling REST APIs with `curl` and processing their responses. By the end you'll be able to treat any JSON-speaking API as a first-class data source in your scripts.

> **Install jq:** `apt install jq` / `brew install jq` / `dnf install jq`. Verify with `jq --version`. All examples in this chapter assume jq 1.6 or later.

1 — jq Basics: Identity, Field Access, and Array Indexing

Every jq program is a *filter* applied to a JSON input. The input flows in from stdin (or a file), passes through the filter, and the result goes to stdout. The simplest filter is `.` — the identity — which pretty-prints the input unchanged.

🔧 **Field access, array indexing, and basic navigation**

```
# Sample JSON for this section:
# {
#   "name": "Alice",
#   "age": 32,
#   "active": true,
#   "scores": [95, 87, 91],
#   "address": { "city": "London", "zip": "EC1A 1BB" },
#   "tags": ["admin", "editor"]
# }

# ── Identity — pretty-print ──────────────────────────────
echo '{"name":"Alice","age":32}' | jq '.'

# ── Object field access ──────────────────────────────────
jq '.name'            data.json    → "Alice"
jq '.age'             data.json    → 32
jq '.address.city'    data.json    → "London"

# ── Strip quotes from string output ──────────────────────
```

```
jq -r '.name'           data.json    → Alice     (no quotes — use in scripts)

# ── Array indexing ──────────────────────────────────────
jq '.scores[0]'         data.json    → 95
jq '.scores[-1]'        data.json    → 91   (last element)
jq '.scores[1:3]'       data.json    → [87, 91]
jq '.tags[]'            data.json    → "admin"
                                       "editor"  (iterator — one per line)

# ── Optional operator — suppress errors for missing keys ──────
jq '.missing?'          data.json    → (nothing — no error)
jq '.missing // "N/A"' data.json    → "N/A"   (alternative operator)
jq '.age // 0'          data.json    → 32

# ── Keys and values ─────────────────────────────────────
jq 'keys'               data.json    → ["active","address","age","name","scores","tags"]
jq 'keys[]'             data.json    → one key per line
jq 'has("age")'         data.json    → true
jq 'type'               data.json    → "object"
jq '.scores | type'     data.json    → "array"
jq '.scores | length'  data.json    → 3

# ── Assign to bash variables ──────────────────────────────
name=$(jq -r '.name' data.json)
city=$(jq -r '.address.city' data.json)
echo "$name lives in $city"
```

2 — Pipe, Comma, and Constructing New JSON

The jq pipe `|` passes the output of one filter as input to the next — exactly like the shell pipe, but within a single jq program. The comma operator `,` runs two filters on the same input and outputs both results. Square brackets `[]` and curly braces `{}` construct new arrays and objects from the current input.

🔧 **Pipe, comma, and building new structures**

```
# ── Pipe: chain filters ─────────────────────────────────
jq '.address | .city'          data.json   → "London"
jq '.scores | .[0]'            data.json   → 95
jq '.scores | length'          data.json   → 3
jq '.scores | add'             data.json   → 273  (sum of array)
jq '.scores | add / length'   data.json   → 91   (average)
jq '.scores | max'             data.json   → 95
```

```
jq '.scores | min'             data.json   → 87
jq '.scores | sort | reverse' data.json   → [95, 91, 87]

# ── Comma: multiple outputs from one input ──────────────────
jq '.name, .age'               data.json
→ "Alice"
  32

# ── Array construction: collect results into an array ────────
jq '[.name, .age]'             data.json   → ["Alice", 32]
jq '[.scores[] | . * 2]'      data.json   → [190, 174, 182]

# ── Object construction: build a new object ─────────────────
jq '{user: .name, location: .address.city}' data.json
→ { "user": "Alice", "location": "London" }

# Shorthand when key name matches field name:
jq '{name, age}'               data.json
→ { "name": "Alice", "age": 32 }

# ── Object and array expansion ──────────────────────────────
# Spread all fields of a sub-object into current:
jq '{name} + .address'         data.json
→ { "name": "Alice", "city": "London", "zip": "EC1A 1BB" }

# ── to_entries / from_entries — iterate over key-value pairs ──
jq '.address | to_entries'     data.json
→ [{"key":"city","value":"London"}, {"key":"zip","value":"EC1A 1BB"}]

jq '.address | to_entries[] | "\(.key)=\(.value)"' data.json
→ "city=London"
  "zip=EC1A 1BB"

# ── String interpolation with \() ──────────────────────────
jq -r '"Hello \(.name), age \(.age)"' data.json
→ Hello Alice, age 32
```

3 — `select`, `map`, and Working with Arrays of Objects

The most common real-world pattern: you have a JSON array of objects (API responses, log entries, config items) and you need to filter, transform, or aggregate it. `select(condition)` keeps items that pass the test; `map(filter)` applies a transformation to every element.

🔧 **Filtering and transforming arrays of objects**

```
# Sample: GitHub-style array of repos
# [
#   {"name":"alpha","stars":142,"lang":"Python","archived":false},
#   {"name":"beta", "stars":38, "lang":"Go",    "archived":false},
#   {"name":"gamma","stars":891,"lang":"Python","archived":true},
#   {"name":"delta","stars":12, "lang":"Rust",  "archived":false}
# ]

# ── select: filter items by condition ──────────────────────
jq '.[] | select(.stars > 100)'         repos.json
→ alpha (142) and gamma (891)

jq '.[] | select(.lang == "Python")'     repos.json
jq '.[] | select(.archived == false)'    repos.json
jq '.[] | select(.archived | not)'       repos.json  # same

# Combine conditions (and = ,  or = |)
jq '.[] | select(.lang == "Python" and .archived == false)' repos.json

# ── Collect filtered results into an array ─────────────────
jq '[.[] | select(.stars > 50)]'         repos.json

# ── map: apply filter to every element ─────────────────────
jq 'map(.name)'                           repos.json
→ ["alpha","beta","gamma","delta"]

jq 'map({name, stars})'                   repos.json
→ [{"name":"alpha","stars":142}, ...]

jq 'map(select(.archived == false)) | map(.name)' repos.json
→ ["alpha","beta","delta"]

# Equivalent shorthand:
jq [.[]|select(.archived|not)|.name]      repos.json

# ── map_values: transform values of an object ──────────────
jq '.address | map_values(ascii_upcase)'  data.json
→ {"city":"LONDON","zip":"EC1A 1BB"}

# ── sort_by, group_by, unique_by ───────────────────────────
jq 'sort_by(.stars)'                      repos.json   # ascending
jq 'sort_by(.stars) | reverse'            repos.json   # descending
jq 'sort_by(.name)'                       repos.json   # alphabetical
```

```
jq 'group_by(.lang)'                    repos.json
→ [[{Go items}], [{Python items}], [{Rust items}]]

jq 'unique_by(.lang) | map(.lang)'      repos.json
→ ["Go","Python","Rust"]

# ── any / all ──────────────────────────────────────────
jq 'any(.[]; .stars > 500)'             repos.json   → true
jq 'all(.[]; .archived == false)'       repos.json   → false

# ── flatten ────────────────────────────────────────────
jq '. | flatten'         <<< '[[1,2],[3,[4,5]]]'   → [1,2,3,4,5]
jq '. | flatten(1)'      <<< '[[1,2],[3,[4,5]]]'   → [1,2,3,[4,5]]
```

4 — Advanced Filters: reduce, Paths, and Shell Variable Injection

Shell variables in jq — the right way

> **Never interpolate shell variables directly into jq filters.** `jq ".name == \"$name\""` breaks on special characters and is a code injection risk. Always pass shell values via `--arg` (string) or `--argjson` (JSON value).

🔧 **Passing shell variables into jq safely**

```
# ── --arg NAME VALUE — injects a shell string as $NAME ──────
lang="Python"
jq --arg lang "$lang" \
   '.[] | select(.lang == $lang) | .name' repos.json

threshold=100
# --arg makes it a string; compare with tonumber to use as int
jq --arg t "$threshold" \
   '.[] | select(.stars > ($t | tonumber))' repos.json

# ── --argjson NAME JSON — injects a real JSON value as $NAME ──
jq --argjson t "$threshold" \
   '.[] | select(.stars > $t)' repos.json   # $t is already int — no conversion

jq --argjson active "false" \
```

```
    '.[] | select(.archived == $active)' repos.json

# ── --slurpfile NAME FILE — reads FILE as JSON array into $NAME
jq --slurpfile cfg config.json '$cfg[0].timeout'

# ── --rawfile NAME FILE — reads FILE as raw string into $NAME ─
jq --rawfile tmpl template.txt '{template: $tmpl}'

# ── env object — access environment variables ─────────────────
jq -n 'env.HOME'                          → "/home/user"
jq -n 'env | keys | length'               → count of env vars
API_KEY="secret123"
jq -n '{"key": env.API_KEY}'              → {"key":"secret123"}

# ── -n flag — null input (no stdin needed) ────────────────────
# Useful when building JSON from scratch or from --arg values
jq -n --arg name "Alice" --argjson age 32 \
   '{name: $name, age: $age}'
→ {"name":"Alice","age":32}
```

reduce and foreach

🔧 **reduce for accumulation, paths for deep navigation**

```
# ── reduce: fold an array into a single value ─────────────────
# reduce EXPR as $var (INIT; ACCUMULATOR)

# Sum of scores:
jq 'reduce .scores[] as $s (0; . + $s)'  data.json    → 273

# Sum of stars across all repos:
jq 'reduce .[] as $r (0; . + $r.stars)'  repos.json   → 1083

# Build a lookup map: name → stars
jq 'reduce .[] as $r ({}; . + {($r.name): $r.stars})' repos.json
→ {"alpha":142,"beta":38,"gamma":891,"delta":12}

# ── paths: navigate deeply nested structures ──────────────────
# paths outputs every path in the document as an array
jq '[paths]'            data.json
→ [["name"],["age"],["address","city"],["address","zip"],...]

# getpath / setpath / delpaths
jq 'getpath(["address","city"])'         data.json   → "London"
```

```
jq 'setpath(["address","country"]; "UK")' data.json
jq 'delpaths([["age"],["scores"]])'       data.json

# ── walk: recursively transform every node ──────────────────
# Lowercase all string values in an entire document:
jq 'walk(if type == "string" then ascii_downcase else . end)' data.json

# ── limit: stop iteration early ─────────────────────────────
jq '[limit(3; .[])]'    repos.json            → first 3 items
jq 'first(.[] | select(.stars > 100))' repos.json  → first match
```

5 — Building JSON from Shell Data

Often you need to go the other direction: take shell variables and system data and assemble them into a JSON payload — for an API call, a log entry, or a config file. `jq -n` with `--arg`/`--argjson` is the safe, composable way to do this.

🔧 **Constructing JSON payloads from shell variables**

```
# ── Simple object from shell variables ──────────────────────
hostname=$(hostname)
uptime_secs=$(awk '{print int($1)}' /proc/uptime)
timestamp=$(date -Is)

jq -n \
  --arg     host    "$hostname"     \
  --argjson uptime  "$uptime_secs"  \
  --arg     ts      "$timestamp"    \
  '{
    "hostname": $host,
    "uptime_seconds": $uptime,
    "timestamp": $ts
  }'

# ── Build a JSON array from a bash array ────────────────────
services=( nginx postgres redis )
# Method 1: printf + jq --slurp --raw-input
printf '%s\n' "${services[@]}" | jq -R '.' | jq -s '.'
→ ["nginx","postgres","redis"]

# Method 2: build directly with --arg and string split
svc_list="nginx,postgres,redis"
```

```
jq -n --arg svcs "$svc_list" '$svcs | split(",")'
→ ["nginx","postgres","redis"]

# ── Collect system metrics into a JSON report ──────────────
system_json() {
    local host cpu_idle mem_free mem_total disk_pct
    host=$(hostname -f)
    cpu_idle=$(top -bn1 | awk '/Cpu/ {print $8+0}')
    mem_free=$(awk '/MemAvailable/{print $2}' /proc/meminfo)
    mem_total=$(awk '/MemTotal/{print $2}' /proc/meminfo)
    disk_pct=$(df / | awk 'NR==2{print $5+0}')

    jq -n \
      --arg     host      "$host"       \
      --argjson cpu_idle  "$cpu_idle"   \
      --argjson mem_free  "$mem_free"   \
      --argjson mem_total "$mem_total"  \
      --argjson disk_pct  "$disk_pct"   \
      --arg     ts        "$(date -Is)" \
      '{
        hostname:       $host,
        timestamp:      $ts,
        cpu_idle_pct:   $cpu_idle,
        memory: {
          total_kb:     $mem_total,
          available_kb: $mem_free,
          used_pct:     (100 - ($mem_free * 100 / $mem_total) | round)
        },
        disk_root_pct:  $disk_pct
      }'
}

system_json | tee metrics.json
```

Updating existing JSON

🔧 The |= update operator and field manipulation

```
# |= updates a field in place (reads current value as .)
jq '.age |= . + 1'                  data.json   # increment age
jq '.name |= ascii_upcase'          data.json   # uppercase name
jq '.scores |= map(. + 5)'          data.json   # add 5 to every score
jq '.tags |= . + ["reviewer"]'      data.json   # append to array
jq '.address.country = "UK"'        data.json   # add new field
```

```
jq 'del(.age)'                          data.json   # remove field
jq 'del(.address.zip)'                 data.json   # remove nested field

# ── Merge two JSON objects ──────────────────────────────
# Shallow merge (right wins on conflict):
jq -s '.[0] * .[1]' base.json override.json

# Deep merge (recursive, right wins):
jq -s '
  def deep_merge(a; b):
    if (a | type) == "object" and (b | type) == "object"
    then reduce (b | keys[]) as $k (a; .[$k] = deep_merge(a[$k]; b[$k]))
    else b
    end;
  deep_merge(.[0]; .[1])
' base.json override.json

# ── Edit a JSON config file in-place ────────────────────
tmp=$(mktemp)
jq '.database.port = 5433' config.json > "$tmp" && mv "$tmp" config.json
```

jq has no in-place edit flag — always redirect to a temp file, then move it over the original. This is atomic on POSIX filesystems (same partition).

6 — Calling REST APIs with `curl` + `jq`

The standard pattern: `curl` fetches the JSON response; `jq` extracts what you need. Topic 8 covers `curl` in depth — here we focus on the processing side and the patterns that make API scripts robust.

🔧 **curl + jq patterns for REST APIs**

```
# ── Basic GET → extract field ──────────────────────────
curl -s https://api.github.com/repos/stedolan/jq \
  | jq -r '.stargazers_count'

# ── POST with JSON body ─────────────────────────────────
payload=$(jq -n --arg user "alice" --arg pass "$PASSWORD" \
          '{username: $user, password: $pass}')

curl -s -X POST \
     -H 'Content-Type: application/json' \
     -d "$payload" \
     https://api.example.com/auth/login \
```

```
  | jq -r '.token'

# ── Check HTTP status + parse body ──────────────────────
response=$(curl -s -w '\n%{http_code}' https://api.example.com/users)
status=$(tail -n1 <<< "$response")
body=$(head -n-1 <<< "$response")

if [[ "$status" != "200" ]]; then
    echo "API error $status: $(jq -r '.message // "unknown"' <<< "$body")" >&2
    exit 1
fi
jq '.' <<< "$body"

# ── Paginated API — collect all pages ──────────────────
fetch_all_pages() {
    local url="$1"
    local page=1
    local all_items='[]'

    while true; do
        local data
        data=$(curl -s "${url}?page=${page}&per_page=100")
        local count; count=$(jq 'length' <<< "$data")
        (( count == 0 )) && break
        all_items=$(jq -s '.[0] + .[1]' <<< "${all_items}"$'\n'"${data}")
        (( count < 100 )) && break
        (( page++ ))
    done
    printf '%s' "$all_items"
}

# ── GitHub: list all repos and their star counts ────────
github_repos() {
    local org="$1"
    local token="${GITHUB_TOKEN:-}"
    local auth_header=""
    [[ -n "$token" ]] && auth_header="-H 'Authorization: token $token'"

    curl -s $auth_header \
         "https://api.github.com/orgs/${org}/repos?per_page=100&sort=stars" \
    | jq -r '.[] | [.name, .stargazers_count, .language // "N/A"] | @tsv'
}
# @tsv formats an array as a tab-separated line — perfect for shell processing
```

Output formats: @base64, @uri, @csv, @tsv, @html, @sh

🔧 **jq format strings for output encoding**

```
# @tsv — format array as TSV row (great for shell consumption)
jq -r '.[] | [.name, .stars, .lang] | @tsv'  repos.json
→ alpha 142     Python
   beta 38      Go

# @csv — format array as CSV row (quoted if needed)
jq -r '.[] | [.name, .stars, .lang] | @csv'  repos.json
→ "alpha",142,"Python"

# @sh — shell-quote values (safe to eval)
jq -r '.name | @sh'                          data.json
→ 'Alice'   (safe even if name contained spaces or quotes)

# @base64 / @base64d
jq -r '"hello world" | @base64'
→ aGVsbG8gd29ybGQ=

jq -r '"aGVsbG8gd29ybGQ=" | @base64d'
→ hello world

# @uri — URL-encode a string
jq -rn '"hello world" | @uri'
→ hello%20world

# @html — HTML-escape
jq -rn '"<b>bold</b> & \" quote" | @html'
→ &lt;b&gt;bold&lt;/b&gt; &amp; " quote

# ── Read @tsv output back into a while loop ──────────────────
while IFS=$'\t' read -r name stars lang; do
    printf '%-15s %5s stars  (%s)\n' "$name" "$stars" "$lang"
done < <(jq -r '.[] | [.name, .stars, .lang] | @tsv' repos.json)
```

7 — Streaming, Multiple Documents, and Compact Output

🔧 **Handling multiple JSON documents and large files**

```
# ── -s / --slurp: read all input into one array ──────────────
# Useful when input is multiple JSON documents (one per line = JSONL)
```

```
jq -s '.' file1.json file2.json    # → [doc1, doc2]
jq -s '.[0] + .[1]' a.json b.json  # → merged array

# ── JSONL (JSON Lines) — one JSON object per line ──────────
# Common in log files, streaming APIs, database exports
# {"ts":"2026-06-09","level":"ERROR","msg":"disk full"}
# {"ts":"2026-06-09","level":"INFO", "msg":"restarted"}

# Process JSONL: each line is a separate document
jq '.level'                          app.jsonl   # prints each level
jq 'select(.level == "ERROR")'        app.jsonl   # filter errors only
jq -r '[.ts, .level, .msg] | @tsv'   app.jsonl   # tabular output

# Count errors by level:
jq -s 'group_by(.level) | map({(.[0].level): length}) | add' app.jsonl

# ── Convert JSON array to JSONL ────────────────────────────
jq -c '.[]'         repos.json     # -c = compact, one object per line

# ── Convert JSONL to JSON array ────────────────────────────
jq -s '.'           app.jsonl

# ── --stream for very large files (no slurp into memory) ──────
# --stream outputs [path,value] pairs, one at a time
jq --stream 'select(.[0][-1] == "name") | .[1]' huge.json

# ── -c compact: no whitespace — for writing back JSON files ──
jq -c '.database.port = 5433' config.json  # compact single-line output

# ── Combine multiple JSON files ────────────────────────────
# Merge all *.json files in a directory into one array
jq -s '.' *.json

# Add a source filename to each document
for f in *.json; do
    jq --arg src "$f" '. + {_source: $src}' "$f"
done | jq -s '.'   # collect into one array
```

8 — Quick Reference

FILTER / FLAG	WHAT IT DOES	EXAMPLE
`.`	Identity / pretty-print	`jq '.' file.json`

FILTER / FLAG	WHAT IT DOES	EXAMPLE
`.key`	Object field access	*.name → "Alice"*
`.a.b`	Nested field	*.address.city*
`.arr[N]`	Array index (negative OK)	*.scores[-1]*
`.arr[]`	Iterate array	*one value per output*
`// default`	Alternative if null/false	*.x // "N/A"*
`|`	Pipe: chain filters	*.arr | length*
`,`	Multiple outputs	*.a, .b*
`[f]`	Collect outputs into array	*[.[] | select(…)]*
`{k: f}`	Build new object	*{name, city: .address.city}*
`select(cond)`	Keep if condition is true	*select(.age > 18)*
`map(f)`	Apply filter to every element	*map(.name)*
`sort_by(f)`	Sort array by key	*sort_by(.stars) | reverse*
`group_by(f)`	Group array by key	*group_by(.lang)*
`unique_by(f)`	Deduplicate by key	
`to_entries`	Object → [{key,value}]	*to_entries[] | .key*
`from_entries`	[{key,value}] → object	
`reduce A as $x (I; E)`	Fold/accumulate	*reduce .[] as $n (0; .+$n)*
`.|= f`	Update in place	*.age |= . + 1*
`del(.k)`	Remove field	
`@tsv @csv @sh @base64 @uri`	Output format encodings	*[.a,.b] | @tsv*
`-r`	Raw string output (no quotes)	
`-c`	Compact output (no whitespace)	
`-n`	Null input (build from scratch)	
`-s`	Slurp all input into array	
`--arg N V`	Inject shell string as $N	
`--argjson N V`	Inject JSON value as $N	
`-e`	Exit 1 if output is null/false	*useful in if statements*

Exercises

All exercises use real-world patterns you'll encounter when scripting against APIs and JSON data stores.

EXERCISE 1

Write a script called `json_report.sh` that reads a JSONL file of server log entries (each line: `{"ts":"…","host":"…","level":"INFO|WARN|ERROR","msg":"…","duration_ms":N}`) and produces a summary report showing: total entries, counts per level, the top 5 slowest requests (host + msg + duration), and average duration per log level. Accept the file path as an argument; default to stdin.

Hint: use `jq -s` to slurp all lines into an array. Use `group_by(.level)` for per-level stats. For top-5 slowest use `sort_by(.duration_ms) | reverse | .[0:5]`. Use `reduce` or `map(.duration_ms) | add / length` for averages. Format the final output with multiple `jq` calls or a single large program with `def`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# json_report.sh [JSONL_FILE]
set -euo pipefail

FILE="${1:--}"   # default to stdin

# Generate test data if no file given and stdin is a terminal
if [[ "$FILE" == "-" && -t 0 ]]; then
    cat <<'EOF'
{"ts":"2026-06-09T10:00:01","host":"web-1","level":"INFO", "msg":"GET /api/users",
{"ts":"2026-06-09T10:00:02","host":"web-2","level":"ERROR","msg":"GET /api/orders",
{"ts":"2026-06-09T10:00:03","host":"web-1","level":"WARN", "msg":"POST /api/upload"
{"ts":"2026-06-09T10:00:04","host":"web-3","level":"INFO", "msg":"GET /api/users",
{"ts":"2026-06-09T10:00:05","host":"web-2","level":"ERROR","msg":"DELETE /api/item"
{"ts":"2026-06-09T10:00:06","host":"web-1","level":"INFO", "msg":"GET /healthz",
{"ts":"2026-06-09T10:00:07","host":"web-3","level":"WARN", "msg":"PUT /api/config",
EOF
    exit 0
fi

# All analysis in one jq program
jq -rs '
  # Summary stats
  . as $all |
```

```
  {
    total: length,
    levels: (group_by(.level) | map({
      level:   .[0].level,
      count:   length,
      avg_ms:  (map(.duration_ms) | add / length | round),
      max_ms:  map(.duration_ms) | max
    })),
    slowest: (sort_by(.duration_ms) | reverse | .[0:5] | map({
      host,
      msg,
      level,
      duration_ms
    })),
    overall_avg_ms: (map(.duration_ms) | add / length | round),
    overall_max_ms: (map(.duration_ms) | max)
  }
' "$FILE" | jq -r '
  "═══════════════════════════════════════",
  "  LOG REPORT SUMMARY",
  "═══════════════════════════════════════",
  "  Total entries:  \(.total)",
  "  Overall avg:    \(.overall_avg_ms)ms",
  "  Overall max:    \(.overall_max_ms)ms",
  "",
  "  BY LEVEL:",
  (.levels[] | "  \(.level)\t count=\(.count)\t avg=\(.avg_ms)ms\t max=\(.max_ms)ms
  "",
  "  TOP 5 SLOWEST REQUESTS:",
  (.slowest[] | "  [\(.level)] \(.host) \(.msg) — \(.duration_ms)ms")
'
```

EXERCISE 2

Write a script called `json_diff.sh` that compares two JSON files and reports: keys present in file 1 but not file 2, keys present in file 2 but not file 1, and keys present in both but with different values. Work at the top level only (no deep recursion required). Accept both file paths as arguments.

Hint: use `jq -s` to load both files into a two-element array. Use `keys` to get key sets, then set operations: `-` (difference) isn't built-in, but you can use `map(select(. as $k | other_keys | contains([$k]) | not))`. For value comparison, iterate keys that appear in both and compare with `.[0][$k] != .[1][$k]`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# json_diff.sh FILE1 FILE2
set -euo pipefail

F1="${1:?Usage: json_diff.sh FILE1 FILE2}"
F2="${2:?file2 required}"

for f in "$F1" "$F2"; do
    [[ -f "$f" ]] || { printf 'Not found: %s\n' "$f" >&2; exit 1; }
done

jq -rs \
  --arg f1 "$F1" \
  --arg f2 "$F2" \
  '
  .[0] as $a | .[1] as $b |
  ($a | keys) as $ka |
  ($b | keys) as $kb |

  # Keys only in file 1
  ($ka | map(select(. as $k | $kb | contains([$k]) | not))) as $only_a |
  # Keys only in file 2
  ($kb | map(select(. as $k | $ka | contains([$k]) | not))) as $only_b |
  # Keys in both but different values
  ($ka | map(select(. as $k | $kb | contains([$k])))  |
   map(select(. as $k | $a[$k] != $b[$k]))) as $changed |

  "Comparing \($f1) vs \($f2)\n",

  if ($only_a | length) > 0 then
    "Only in \($f1):",
    ($only_a[] | "  - \(.)  = \($a[.])")
  else "Only in \($f1): (none)" end,
  "",

  if ($only_b | length) > 0 then
    "Only in \($f2):",
    ($only_b[] | "  + \(.)  = \($b[.])")
```

```
  else "Only in \($f2): (none)" end,
  "",

  if ($changed | length) > 0 then
    "Changed values:",
    ($changed[] | "  ~ \(.)\n      was: \($a[.])\n      now: \($b[.])")
  else "Changed values: (none)" end
' "$F1" "$F2"
```

EXERCISE 3

Write a function called `gh_repo_stats()` that calls the GitHub API to fetch statistics about a given `owner/repo`: stars, forks, open issues, language, last push date, and the 5 most recent open issues (title + created_at + labels). If `GITHUB_TOKEN` is set in the environment, use it for authentication. Format the output as a readable summary. Handle API errors gracefully — if the repo doesn't exist or the rate limit is hit, print a helpful message to stderr and return 1.

Hint: make two curl calls — one to `/repos/owner/repo` and one to `/repos/owner/repo/issues?state=open&per_page=5&sort=created`. Check the HTTP status code with `-w '%{http_code}'`. For the issues list, use `jq -r '.[] | [.number, .title, .created_at, ([.labels[].name] | join(","))] | @tsv'`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
gh_repo_stats() {
    local repo="${1:?Usage: gh_repo_stats owner/repo}"
    local api="https://api.github.com"
    local auth=()
    [[ -n "${GITHUB_TOKEN:-}" ]] && \
        auth=( -H "Authorization: Bearer $GITHUB_TOKEN" )

    _gh_get() {
        local url="$1"
        local out
        out=$(curl -s -w '\n%{http_code}' \
                   -H 'Accept: application/vnd.github+json' \
                   "${auth[@]}" "$url")
        local status="${out##*$'\n'}"
        local body="${out%$'\n'*}"

        if [[ "$status" == "404" ]]; then
```

```
            printf 'Error: repository "%s" not found\n' "$repo" >&2
            return 1
        elif [[ "$status" == "403" ]]; then
            local msg; msg=$(jq -r '.message' <<< "$body")
            printf 'Error 403: %s\n' "$msg" >&2
            return 1
        elif [[ "$status" != "200" ]]; then
            printf 'HTTP error %s for %s\n' "$status" "$url" >&2
            return 1
        fi
        printf '%s' "$body"
    }

    # Fetch repo metadata
    local info; info=$(_gh_get "${api}/repos/${repo}") || return 1

    # Fetch recent open issues
    local issues; issues=$(_gh_get \
        "${api}/repos/${repo}/issues?state=open&per_page=5&sort=created&direction=d
    ) || return 1

    # Print summary
    jq -r '
      "╔══════════════════════════════════════════╗",
      "║  " + .full_name + " "*( 40 - (.full_name|length)) + "║",
      "╚══════════════════════════════════════════╝",
      "  Stars:       \(.stargazers_count)",
      "  Forks:       \(.forks_count)",
      "  Open issues: \(.open_issues_count)",
      "  Language:    \(.language // "N/A")",
      "  Last push:   \(.pushed_at | split("T")[0])",
      "  Description: \(.description // "(none)")"
    ' <<< "$info"

    printf '\n  RECENT OPEN ISSUES:\n'
    jq -r '
      if length == 0 then "  (none)"
      else
        .[] | "  #\(.number) \(.title)\n        Created: \(.created_at | split("T")
      end
    ' <<< "$issues"
}
```

```
# Usage:
# gh_repo_stats stedolan/jq
# GITHUB_TOKEN=ghp_... gh_repo_stats PhilipOsztromok/learning_blog
```

EXERCISE 4

Write a script called `json_to_env.sh` that takes a JSON object (from a file or stdin) and outputs its key-value pairs as shell export statements, suitable for use with `eval` or `source`. Nested objects should be flattened with underscore-joined key names (e.g. `{"db":{"host":"localhost"}}` → `export DB_HOST='localhost'`). Arrays should be joined with commas. Keys should be uppercased. Handle values that contain single quotes safely.

Hint: use `paths(scalars)` to get all leaf paths, then `getpath(path)` for each value. Join the path components with `_` and uppercase with `ascii_upcase`. For arrays, check `type == "array"` at the path's parent and use `join(",")`. Use `@sh` format to safely quote the value. ▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# json_to_env.sh [FILE]  — or pipe JSON on stdin
# Outputs: export KEY='value' lines for all scalar leaf values
set -euo pipefail

INPUT="${1:--}"

jq -r '
  # Recurse through all paths, emitting key=value for each scalar leaf
  # and for each array (join with comma)
  def flatten_paths:
    paths(scalars),
    paths(arrays | length > 0 | not | not)  # non-empty arrays
    | . ;

  # Visit every scalar and array leaf
  [paths(scalars), paths(arrays)] | unique[] as $p |
  . as $root |
  ($p | map(tostring) | join("_") | ascii_upcase) as $key |
  ($root | getpath($p)) as $val |
  if ($val | type) == "array" then
    $key + "=" + ($val | map(tostring) | join(",") | @sh)
  else
    $key + "=" + ($val | tostring | @sh)
  end |
  "export " + .
' "$INPUT"
```

```
# Example input:
# {
#   "db": {"host": "localhost", "port": 5432, "name": "myapp"},
#   "app": {"debug": false, "workers": 4},
#   "allowed_hosts": ["example.com", "api.example.com"]
# }
#
# Output:
# export ALLOWED_HOSTS='example.com,api.example.com'
# export APP_DEBUG='false'
# export APP_WORKERS='4'
# export DB_HOST='localhost'
# export DB_NAME='myapp'
# export DB_PORT='5432'
#
# Usage:
# eval "$(./json_to_env.sh config.json)"
# echo "$DB_HOST"  → localhost
```

🌐 Intermediate Topic 8 — Network Operations from the Shell

Shell scripts are first-class network citizens. `curl` alone covers the vast majority of HTTP work: downloading files, calling REST APIs, sending webhooks, checking service health, and posting alerts. Below HTTP, bash's built-in `/dev/tcp` pseudo-device handles raw TCP without any external tool. This chapter covers `curl` in production depth — headers, authentication, retries, rate limiting, parallel downloads — along with `wget` for mirroring, `/dev/tcp` for low-level probing, and the patterns used in real monitoring and deployment scripts.

1 — `curl` In Depth

`curl` is the workhorse. The flags you use every day are only a fraction of what it offers. Understanding its output model — where the body goes, where headers go, how to capture the HTTP status code — is what separates scripts that work from scripts that silently fail.

🌐 Core curl flags and output model

```
# ── Output control ──────────────────────────────────────
curl -s   URL           # silent: suppress progress/errors
curl -S   URL           # show errors even when silent (-sS together)
```

```
curl -o FILE URL        # write body to FILE instead of stdout
curl -O   URL           # write to filename from URL
curl -L   URL           # follow redirects (3xx)
curl -I   URL           # HEAD only — show response headers, no body
curl -v   URL           # verbose: show all request and response headers

# ── Capture HTTP status code separately ──────────────────────
# -w writes a format string AFTER the response body
http_code=$(curl -sS -o /dev/null -w '%{http_code}' URL)

# Body + status in one call — split on the last newline
response=$(curl -sS -w $'\n%{http_code}' URL)
status="${response##*$'\n'}"
body="${response%$'\n'*}"

# ── Useful -w variables ──────────────────────────────────────
# %{http_code}      HTTP status (200, 404, etc.)
# %{time_total}     total elapsed seconds
# %{time_connect}   time to TCP connect
# %{size_download}  bytes downloaded
# %{speed_download} bytes/second
# %{redirect_url}   final URL after redirects
# %{content_type}   Content-Type header value

curl -sS -o /dev/null \
     -w 'status=%{http_code}  time=%{time_total}s  size=%{size_download}B\n' \
     https://example.com

# ── Timeouts ─────────────────────────────────────────────────
curl --connect-timeout 5   # max seconds to establish TCP connection
curl --max-time       30   # max total seconds for the whole transfer
# Always set both — without them curl hangs indefinitely

# ── Request methods and bodies ───────────────────────────────
curl -X GET    URL
curl -X POST   -d 'key=val'               URL
curl -X POST   -d '{"x":1}' \
               -H 'Content-Type: application/json'  URL
curl -X PUT    -d @payload.json \
               -H 'Content-Type: application/json'  URL
curl -X DELETE URL
curl -X PATCH  -d '{"field":"value"}' \
               -H 'Content-Type: application/json'  URL

# ── Headers ──────────────────────────────────────────────────
curl -H 'Accept: application/json'          URL
curl -H 'Authorization: Bearer TOKEN'       URL
```

```
curl -H 'X-API-Key: SECRET'                  URL
curl -H 'Authorization: Basic BASE64CREDS'   URL
# Or let curl encode basic auth:
curl -u 'user:password'                       URL

# ── Dump response headers to a file ──────────────────────────
curl -sSD headers.txt -o body.json URL
# -D FILE writes response headers; -o FILE writes body

# ── SSL / TLS ────────────────────────────────────────────────
curl -k              URL   # skip cert verification (dev only)
curl --cacert ca.pem  URL   # custom CA bundle
curl --cert cl.pem \
     --key  cl-key.pem URL  # mutual TLS (client cert)
```

Retries and rate limiting

curl built-in retry and production-safe wrappers

```
# ── Built-in retry flags (curl 7.12+) ────────────────────────
curl --retry           3    # retry up to N times on transient failures
curl --retry-delay     2    # seconds between retries
curl --retry-max-time  60   # total retry budget in seconds
curl --retry-connrefused    # also retry on connection refused
# curl retries on: network errors, 408, 429, 500, 502, 503, 504
# curl does NOT retry on 4xx errors by default

# ── Production-safe curl wrapper ─────────────────────────────
api_call() {
    # Usage: api_call METHOD URL [extra curl args...]
    local method="$1" url="$2"; shift 2
    local response status body

    response=$(curl -sS                   \
                    -X "$method"           \
                    --connect-timeout 10   \
                    --max-time       30    \
                    --retry          3     \
                    --retry-delay    2     \
                    --retry-connrefused    \
                    -w $'\n%{http_code}'   \
                    "$@"                   \
                    "$url")
```

```
    status="${response##*$'\n'}"
    body="${response%$'\n'*}"

    if [[ "$status" -ge 200 && "$status" -lt 300 ]]; then
        printf '%s' "$body"
        return 0
    fi

    printf 'HTTP %s from %s: %s\n' "$status" "$url" \
           "$(jq -r '.message // .error // "no message"' <<< "$body" 2>/dev/null || printf '
    return 1
}

# Usage:
users=$(api_call GET https://api.example.com/users \
         -H "Authorization: Bearer $TOKEN")

api_call POST https://api.example.com/items \
         -H 'Content-Type: application/json' \
         -d '{"name":"widget"}'

# ── Manual rate limiting (honour 429 Retry-After) ──────────
curl_rate_limited() {
    local url="$1"; shift
    local attempt

    for (( attempt=1; attempt<=5; attempt++ )); do
        local headers body status
        local tmphead; tmphead=$(mktemp)

        body=$(curl -sS -w $'\n%{http_code}' -D "$tmphead" "$@" "$url")
        status="${body##*$'\n'}"
        body="${body%$'\n'*}"

        if [[ "$status" == "429" ]]; then
            local retry_after
            retry_after=$(grep -i 'Retry-After:' "$tmphead" | \
                          awk '{print $2}' | tr -d $'\r')
            retry_after=${retry_after:-5}
            printf 'Rate limited — sleeping %ds (attempt %d/5)\n' \
                   "$retry_after" "$attempt" >&2
            sleep "$retry_after"
            rm -f "$tmphead"
            continue
        fi

        rm -f "$tmphead"
```

```
        printf '%s' "$body"
        [[ "$status" -ge 200 && "$status" -lt 300 ]]
        return
    done

    printf 'Rate limit exceeded after 5 attempts\n' >&2
    return 1
}
```

File transfer and parallel downloads

🌐 **Downloading files, progress display, parallel fetches**

```
# ── Download with progress bar ──────────────────────────
curl -L --progress-bar -o output.tar.gz URL

# ── Resume an interrupted download ──────────────────────
curl -C - -o partial.tar.gz URL   # -C - = auto-detect resume offset

# ── Download only if remote is newer (conditional GET) ──
curl -z existing_file.txt -o existing_file.txt URL
# Sends If-Modified-Since based on local file's mtime

# ── Parallel downloads with xargs ───────────────────────
# Download 4 URLs concurrently
printf '%s\n' "${urls[@]}" | \
    xargs -P4 -I{} curl -sS -L -o '{}'.html '{}'

# ── Upload a file ───────────────────────────────────────
curl -F "file=@/path/to/upload.txt" URL          # multipart form
curl -T /path/to/file ftp://server/dest/         # FTP PUT
curl -X PUT --data-binary @file.bin URL          # binary body

# ── Send form data (URL-encoded) ────────────────────────
curl -d 'user=alice&pass=secret' URL
curl --data-urlencode 'msg=hello world' URL   # auto URL-encodes spaces

# ── Cookies ─────────────────────────────────────────────
curl -c cookies.txt -b cookies.txt URL   # save + send cookies

# ── Use a config file to avoid repeating flags ──────────
# ~/.curlrc or a project-specific file
cat > .curlrc <<'EOF'
```

```
silent
show-error
location
connect-timeout = 10
max-time = 60
retry = 3
EOF
curl --config .curlrc URL
```

2 — HTTP Health Checks and Service Monitoring

🌐 **Health check patterns used in production scripts**

```bash
# ── Simple: is the service up? ──────────────────────────
http_ok() {
    local url="$1"
    local code
    code=$(curl -sS -o /dev/null -w '%{http_code}' \
              --connect-timeout 5 --max-time 10 "$url" 2>/dev/null)
    [[ "$code" -ge 200 && "$code" -lt 400 ]]
}

http_ok https://api.example.com/health && echo "up" || echo "down"

# ── Check multiple services and report status ───────────
check_services() {
    local -a endpoints=( "$@" )
    local ok=0 failed=0

    printf '%-45s %-6s %s\n' "ENDPOINT" "CODE" "TIME"
    printf '%.0s─' {1..65}; echo

    local url
    for url in "${endpoints[@]}"; do
        local result
        result=$(curl -sS -o /dev/null \
                      --connect-timeout 5 --max-time 10 \
                      -w '%{http_code} %{time_total}' \
                      "$url" 2>/dev/null \
                      || echo "000 0")
        local code="${result% *}"
        local time="${result#* }"
```

```
        local colour='\033[32m'   # green
        if   [[ "$code" == "000" ]]; then colour='\033[31m'; (( failed++ ))  # red
        elif (( code >= 400 ));      then colour='\033[31m'; (( failed++ ))
        elif (( code >= 300 ));      then colour='\033[33m'; (( ok++ ))     # yellow
        else                              (( ok++ ))
        fi

        printf "%-45s ${colour}%-6s\033[0m %ss\n" \
               "$url" "$code" "$time"
    done

    printf '\n%d OK, %d failed\n' "$ok" "$failed"
    (( failed == 0 ))
}

check_services \
    https://api.example.com/health \
    https://app.example.com/ping \
    https://db.example.com:8080/status

# ── Wait until a service becomes healthy (deploy pattern) ──────
wait_for_http() {
    local url="$1" timeout="${2:-60}"
    local deadline=$(( $(date '+%s') + timeout ))

    printf 'Waiting for %s (timeout: %ds)...' "$url" "$timeout"
    while (( $(date '+%s') < deadline )); do
        if http_ok "$url"; then
            printf ' ready\n'; return 0
        fi
        printf '.'
        sleep 2
    done
    printf ' TIMEOUT\n' >&2
    return 1
}

wait_for_http http://localhost:8080/health 120
echo "Service is up — starting smoke tests"
```

3 — Sending Notifications via HTTP APIs

🌐 Webhook notifications: Slack, Teams, generic HTTP

```
# ── Slack webhook ──────────────────────────────────────────
slack_notify() {
    local webhook="${SLACK_WEBHOOK_URL:?SLACK_WEBHOOK_URL not set}"
    local message="$1"
    local colour="${2:-good}"   # good | warning | danger

    local payload
    payload=$(jq -n \
        --arg msg    "$message" \
        --arg colour "$colour" \
        '{attachments: [{text: $msg, color: $colour, mrkdwn_in: ["text"]}]}')

    curl -sS -X POST \
         -H 'Content-Type: application/json' \
         --connect-timeout 5 --max-time 15 \
         -d "$payload" "$webhook" > /dev/null
}

# ── PagerDuty / generic alert ──────────────────────────────
pagerduty_alert() {
    local routing_key="${PD_ROUTING_KEY:?}"
    local summary="$1"
    local severity="${2:-error}"   # critical|error|warning|info

    local payload
    payload=$(jq -n \
        --arg key  "$routing_key" \
        --arg sum  "$summary"     \
        --arg sev  "$severity"    \
        --arg src  "$(hostname)"  \
        '{
          routing_key:  $key,
          event_action: "trigger",
          payload: {
            summary:  $sum,
            severity: $sev,
            source:   $src
          }
        }')

    curl -sS -X POST \
         -H 'Content-Type: application/json' \
         -d "$payload" \
```

```
          https://events.pagerduty.com/v2/enqueue
}

# ── Generic webhook dispatcher ──────────────────────────
send_webhook() {
    local url="$1" payload="$2"
    local secret="${WEBHOOK_SECRET:-}"
    local extra_headers=()

    if [[ -n "$secret" ]]; then
        # HMAC-SHA256 signature (GitHub-style)
        local sig
        sig="sha256=$(printf '%s' "$payload" | openssl dgst -sha256 -hmac "$secret" -hex | a
        extra_headers=( -H "X-Hub-Signature-256: $sig" )
    fi

    curl -sS -X POST \
         -H 'Content-Type: application/json' \
         "${extra_headers[@]}" \
         --connect-timeout 5 --max-time 15 \
         -d "$payload" "$url"
}
```

4 — /dev/tcp: Raw TCP Connections from Bash

Bash has a built-in virtual filesystem under `/dev/tcp/HOST/PORT` and `/dev/udp/HOST/PORT`. Opening a file descriptor to these paths establishes a TCP (or UDP) connection — no external tools needed. This is invaluable when you're on a minimal system where `curl`, `netcat`, and `telnet` aren't available.

/dev/tcp for port checks, HTTP, and raw protocol probes

```
# ── TCP port reachability check ─────────────────────────
tcp_port_open() {
    local host="$1" port="$2" timeout="${3:-5}"
    # timeout command wraps the redirect to enforce deadline
    timeout "$timeout" bash -c \
        ">/dev/tcp/${host}/${port}" 2>/dev/null
}

tcp_port_open db.example.com 5432  && echo "DB reachable" || echo "DB unreachable"
tcp_port_open redis.internal  6379  && echo "Redis up"
```

```bash
# ── Raw HTTP request via /dev/tcp ──────────────────────────
http_get_raw() {
    local host="$1" path="${2:-/}" port="${3:-80}"
    exec 3<>/dev/tcp/"${host}"/"${port}"
    printf 'GET %s HTTP/1.0\r\nHost: %s\r\nConnection: close\r\n\r\n' \
           "$path" "$host" >&3
    cat <&3
    exec 3>&-
}

# Use it like curl on a system without curl:
http_get_raw example.com / 80 | head -20

# ── Check if a service speaks a specific protocol ──────────
smtp_banner() {
    local host="$1" port="${2:-25}"
    exec 3<>/dev/tcp/"${host}"/"${port}"
    read -r -t 5 banner <&3
    exec 3>&-
    printf '%s\n' "$banner"
}
smtp_banner mail.example.com     → 220 mail.example.com ESMTP Postfix

# ── Check multiple ports in a loop ─────────────────────────
scan_ports() {
    local host="$1"; shift
    local port
    printf 'Port scan: %s\n' "$host"
    for port in "$@"; do
        if tcp_port_open "$host" "$port" 2; then
            printf '  %-6s \033[32mOPEN\033[0m\n'  "$port"
        else
            printf '  %-6s \033[31mCLOSED\033[0m\n' "$port"
        fi
    done
}

scan_ports db.internal 22 80 443 5432 6379 27017
```

`/dev/tcp` is a bash-only feature — it doesn't exist as a real filesystem entry; the kernel never sees it. It's handled entirely by bash's redirection code. It won't work in `sh`, `dash`, or `zsh` without bash.

5 — wget for Mirroring and Recursive Downloads

`wget` and `curl` overlap heavily for simple downloads, but `wget` has built-in recursive site mirroring and a simpler interface for batch operations. It also handles retries more transparently for download-focused use cases.

wget key patterns

```
# ── Basic download ──────────────────────────────────────
wget URL                      # download to current dir, show progress
wget -q   URL                  # quiet
wget -O outfile URL            # write to specific file
wget -P /dest/ URL             # write to directory

# ── Resume and retry ────────────────────────────────────
wget -c   URL                  # continue/resume partial download
wget --tries=5 URL             # retry up to 5 times
wget --wait=2 URL              # wait 2s between retries
wget --timeout=30 URL          # 30s total timeout

# ── Check existence without downloading body ────────────
wget -q --spider URL           # exit 0 if URL exists, 1 if not
wget -q --spider URL 2>/dev/null && echo "exists"

# ── Mirror a website ────────────────────────────────────
wget --mirror \
     --convert-links \         # rewrite links for local viewing
     --adjust-extension \      # add .html to pages
     --page-requisites \       # also download CSS/JS/images
     --no-parent \             # don't go up to parent dirs
     -P ./mirror/ URL

# ── Download a list of URLs from a file ─────────────────
wget -i urls.txt -P downloads/

# ── wget vs curl: when to use which ─────────────────────
# wget: recursive, mirroring, simple batch downloads
# curl: APIs, custom headers/methods, piping, complex auth
```

6 — Quick Reference

TASK	COMMAND
GET, body to stdout	`curl -sS URL`

TASK	COMMAND
GET, save to file	`curl -sS -L -o file URL`
GET, just the HTTP status	`curl -sS -o /dev/null -w '%{http_code}' URL`
POST JSON	`curl -sS -X POST -H 'Content-Type: application/json' -d '{}' URL`
POST JSON from file	`curl -sS -X POST -H 'Content-Type: application/json' -d @file.json URL`
Bearer token auth	`curl -H "Authorization: Bearer $TOKEN" URL`
Basic auth	`curl -u user:pass URL`
Follow redirects	`curl -L URL`
Set both timeouts	`curl --connect-timeout 5 --max-time 30 URL`
Retry 3 times	`curl --retry 3 --retry-delay 2 URL`
Response headers to file	`curl -D headers.txt -o body.json URL`
Resume download	`curl -C - -o file URL`
TCP port reachable?	`timeout 5 bash -c ">/dev/tcp/host/port"`
Raw GET via /dev/tcp	`exec 3<>/dev/tcp/host/80; printf 'GET / HTTP/1.0\r\n\r\n' >&3; cat <&3`
Check URL exists	`wget -q --spider URL`
Download list of URLs	`wget -i urls.txt -P dest/`
Mirror a site	`wget --mirror --convert-links --page-requisites URL`

📝 Exercises

EXERCISE 1

Write a script called `http_monitor.sh` that reads a list of URLs from a file (one per line, lines starting with `#` are comments), checks each URL every N seconds (configurable via `INTERVAL`, default 60), and logs to a TSV file with columns: timestamp, URL, HTTP status, response time in ms, and UP/DOWN status. When a URL changes state (UP→DOWN or DOWN→UP) it should print a highlighted alert to stderr. Run until interrupted with Ctrl-C, handling the SIGINT cleanly.

Hint: maintain an associative array of last-known states. Use `curl -w '%{http_code} %{time_total}'`. Convert float seconds to ms with integer arithmetic after stripping the decimal. In the SIGINT trap, print a summary of how long each URL was down.

▶ Show Sample Solution

SAMPLE SOLUTION

```bash
#!/usr/bin/env bash
# http_monitor.sh URL_FILE [LOG_FILE]
set -euo pipefail

URL_FILE="${1:?Usage: http_monitor.sh URL_FILE [LOG_FILE]}"
LOG_FILE="${2:-http_monitor.tsv}"
INTERVAL=${INTERVAL:-60}

[[ -f "$URL_FILE" ]] || { printf 'Not found: %s\n' "$URL_FILE" >&2; exit 1; }

declare -A last_state=()
declare -A down_since=()
start_time=$(date '+%s')

# Write TSV header if file is new
[[ -f "$LOG_FILE" ]] || \
    printf 'timestamp\turl\thttp_code\tms\tstatus\n' > "$LOG_FILE"

check_url() {
    local url="$1"
    local ts; printf -v ts '%(%Y-%m-%d %H:%M:%S)T' -1
    local result
    result=$(curl -sS -o /dev/null \
                  --connect-timeout 5 --max-time 15 \
                  -w '%{http_code} %{time_total}' \
                  "$url" 2>/dev/null \
                  || echo "000 0")

    local code="${result% *}"
    local secs="${result#* }"
    # Convert float secs to int ms (remove decimal point, trim leading zeros)
    local ms; ms=$(printf '%.0f' "$(echo "$secs * 1000" | bc)")

    local status
    (( code >= 200 && code < 400 )) && status="UP" || status="DOWN"

    # Log to TSV
    printf '%s\t%s\t%s\t%s\t%s\n' \
           "$ts" "$url" "$code" "$ms" "$status" >> "$LOG_FILE"
```

```
    # State change detection
    local prev="${last_state[$url]:-UNKNOWN}"
    if [[ "$prev" != "$status" ]]; then
        if [[ "$status" == "DOWN" ]]; then
            printf '\033[31m[%s] DOWN: %s (HTTP %s)\033[0m\n' "$ts" "$url" "$code"
            down_since["$url"]=$(date '+%s')
        else
            local outage=""
            [[ -v "down_since[$url]" ]] && \
                outage=" (was down for $(( $(date '+%s') - down_since[$url] ))s)"
            printf '\033[32m[%s] UP: %s%s\033[0m\n' \
                   "$ts" "$url" "$outage" >&2
            unset "down_since[$url]"
        fi
    fi
    last_state["$url"]="$status"
}

# Graceful shutdown
trap '
    printf "\n\033[1mMonitor stopped. Summary:\033[0m\n" >&2
    for u in "${!last_state[@]}"; do
        printf "  %-40s %s\n" "$u" "${last_state[$u]}" >&2
    done
    printf "Runtime: %ds\n" "$(( $(date +%s) - start_time ))" >&2
    exit 0
' INT TERM

# Main loop
printf 'Monitoring %s, interval=%ds, log=%s\n' "$URL_FILE" "$INTERVAL" "$LOG_FILE"
while true; do
    while IFS= read -r url; do
        [[ "$url" =~ ^[[:space:]]*($|#) ]] && continue
        check_url "$url"
    done < "$URL_FILE"
    sleep "$INTERVAL"
done
```

EXERCISE 2

Write a function called `download_with_verify()` that downloads a file from a URL and verifies its integrity against a known checksum. It should accept the URL, the expected checksum, and the hash algorithm (sha256 or sha512) as arguments, write the file to a configurable `DOWNLOAD_DIR`, skip the download if the file already exists and matches the checksum, and clean up a partial download on failure. Print a clear success/failure message.

Hint: use `curl -L -o "$dest"` for the download. Use `sha256sum` or `sha512sum` to compute the hash of the downloaded file. Compare the first field of their output to the expected checksum. Use a trap inside the function to remove the partial file if interrupted.

▶ Show Sample Solution

SAMPLE SOLUTION

```
download_with_verify() {
    local url="${1:?url required}"
    local expected="${2:?expected checksum required}"
    local algo="${3:-sha256}"
    local destdir="${DOWNLOAD_DIR:-.}"

    local filename="${url##*/}"
    local dest="${destdir}/${filename}"

    local sum_cmd
    case "$algo" in
        sha256) sum_cmd="sha256sum" ;;
        sha512) sum_cmd="sha512sum" ;;
        md5)    sum_cmd="md5sum"    ;;
        *) printf 'Unknown algo: %s\n' "$algo" >&2; return 1 ;;
    esac

    _verify() {
        local actual
        actual=$("$sum_cmd" "$dest" | awk '{print $1}')
        [[ "$actual" == "$expected" ]]
    }

    if [[ -f "$dest" ]]; then
        if _verify; then
            printf '[skip] %s already exists and checksum matches\n' "$filename"
            return 0
        fi
        printf '[warn] %s exists but checksum mismatch — re-downloading\n' "$filena
```

```
        rm -f "$dest"
    fi

    mkdir -p "$destdir"

    # Cleanup partial download on any failure
    trap "rm -f '$dest'; printf '\033[31m[fail]\033[0m Download interrupted: %s\n'
         ERR RETURN INT

    printf '[down] %s → %s\n' "$url" "$dest"
    curl -fsSL \
         --connect-timeout 10 \
         --max-time 300       \
         --retry 3            \
         --retry-delay 2      \
         -o "$dest"           \
         "$url"

    if _verify; then
        printf '\033[32m[ok]\033[0m   %s (%s verified)\n' "$filename" "$algo"
        trap - ERR RETURN INT
    else
        local actual; actual=$("$sum_cmd" "$dest" | awk '{print $1}')
        printf '[fail] Checksum mismatch for %s\n  expected: %s\n  got:      %s\n'
               "$filename" "$expected" "$actual" >&2
        rm -f "$dest"
        return 1
    fi
}

# Example usage:
DOWNLOAD_DIR=/tmp/downloads \
download_with_verify \
    https://github.com/stedolan/jq/releases/download/jq-1.6/jq-linux64 \
    af986793a515d500ab2d35f8d2aecd656e764504d683bd632d62991dc4f8eca0  \
    sha256
```

EXERCISE 3

Write a script called `tcp_probe.sh` that acts as a connectivity diagnostic tool. It should accept a host as its argument and then test: DNS resolution (using `getent hosts` or `dig`), TCP connectivity to a

configurable list of ports (from `PORTS` env var, default `22 80 443`), and an HTTP health check if port 80 or 443 is open. Use `/dev/tcp` for the port checks and output a structured report. All checks should have a configurable timeout defaulting to 3 seconds.

Hint: run DNS check first — if that fails, the port checks will also fail and you can report that. For each port check use `timeout $TIMEOUT bash -c ">/dev/tcp/$host/$port"`. For HTTP, do a `/dev/tcp` GET and check for an HTTP response line. Use `SECONDS` or `date +%s%N` to measure each check's latency.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# tcp_probe.sh HOST
set -uo pipefail

HOST="${1:?Usage: tcp_probe.sh HOST}"
PORTS=( ${PORTS:-22 80 443} )
TIMEOUT=${TIMEOUT:-3}

ms_now() { date '+%s%N'; }

elapsed_ms() {
    local t0=$1
    printf '%d' $(( ($(ms_now) - t0) / 1000000 ))
}

print_result() {
    local label="$1" ok="$2" detail="${3:-}"
    if [[ "$ok" == "0" ]]; then
        printf '  \033[32m✓\033[0m %-25s %s\n' "$label" "$detail"
    else
        printf '  \033[31m✗\033[0m %-25s %s\n' "$label" "$detail"
    fi
}

printf '\033[1mTCP Probe: %s\033[0m\n\n' "$HOST"
printf 'DNS RESOLUTION\n'

# ── DNS check ──────────────────────────────────────────
local t0; t0=$(ms_now)
local ip
if ip=$(getent hosts "$HOST" 2>/dev/null | awk '{print $1; exit}') && [[ -n "$ip" ]
```

```
    print_result "DNS" 0 "$ip ($(elapsed_ms $t0)ms)"
else
    print_result "DNS" 1 "FAILED — cannot resolve $HOST"
    printf '\nAll port checks skipped (DNS failed).\n'
    exit 1
fi

# ── Port checks ──────────────────────────────────────────
printf '\nPORT CHECKS\n'
local open_ports=()

for port in "${PORTS[@]}"; do
    t0=$(ms_now)
    if timeout "$TIMEOUT" bash -c \
           ">/dev/tcp/${HOST}/${port}" 2>/dev/null; then
        print_result "Port $port" 0 "OPEN ($(elapsed_ms $t0)ms)"
        open_ports+=( "$port" )
    else
        print_result "Port $port" 1 "CLOSED or filtered"
    fi
done

# ── HTTP check if 80 or 443 is open ──────────────────────
local http_port=""
for p in "${open_ports[@]}"; do
    [[ "$p" == "80" || "$p" == "8080" ]] && { http_port="$p"; break; }
done

if [[ -n "$http_port" ]]; then
    printf '\nHTTP PROBE (port %s)\n' "$http_port"
    t0=$(ms_now)
    local banner=""
    exec 7<>/dev/tcp/"${HOST}"/"${http_port}"
    printf 'HEAD / HTTP/1.0\r\nHost: %s\r\nConnection: close\r\n\r\n' \
           "$HOST" >&7
    if read -r -t "$TIMEOUT" banner <&7; then
        print_result "HTTP response" 0 "${banner//$'\r'/} ($(elapsed_ms $t0)ms)"
    else
        print_result "HTTP response" 1 "No response within ${TIMEOUT}s"
    fi
    exec 7>&-
elif for p in "${open_ports[@]}"; do [[ "$p" == "443" ]] && break; done; then
    printf '\nHTTP PROBE (443 — use curl for HTTPS)\n'
```

```
    t0=$(ms_now)
    local code
    code=$(curl -sS -o /dev/null -w '%{http_code}' \
              --connect-timeout "$TIMEOUT" --max-time "$((TIMEOUT*2))" \
              "https://${HOST}/" 2>/dev/null || echo "000")
    print_result "HTTPS" $(( code < 400 ? 0 : 1 )) "HTTP $code ($(elapsed_ms $t0)ms
fi

printf '\n'
```

EXERCISE 4

Write a script called `api_poll.sh` that polls a JSON API endpoint on a configurable interval, compares the response to the previous one, and logs changes. Usage: `api_poll.sh URL FIELD` where `FIELD` is a jq expression (e.g. `.status` or `.data.count`). When the value of `FIELD` changes, print a timestamped change record to stdout and optionally call a webhook (`ALERT_WEBHOOK` env var). After 10 consecutive failed requests, exit with an error.

Hint: store the previous value in a variable, initially empty (treat first run as baseline). Use `jq -r "$FIELD"` to extract the field. Track consecutive failures with a counter that resets on success. For the webhook, use `curl -X POST -d "$payload"` with a JSON body built by `jq -n`.

▶ Show Sample Solution

SAMPLE SOLUTION

```
#!/usr/bin/env bash
# api_poll.sh URL JQ_FIELD
set -euo pipefail

URL="${1:?Usage: api_poll.sh URL JQ_FIELD}"
FIELD="${2:?jq field expression required}"
INTERVAL=${INTERVAL:-30}
ALERT_WEBHOOK="${ALERT_WEBHOOK:-}"

prev_value=""
fail_count=0
readonly MAX_FAILS=10

printf 'Polling %s field=%s interval=%ds\n' "$URL" "$FIELD" "$INTERVAL" >&2
printf 'timestamp\told_value\tnew_value\n'  # TSV header
```

```
trap 'printf "\nStopped.\n" >&2; exit 0' INT TERM

while true; do
    local body status
    body=$(curl -sS -w $'\n%{http_code}' \
                --connect-timeout 5 --max-time 15 \
                "$URL" 2>/dev/null || echo $'null\n000')
    status="${body##*$'\n'}"
    body="${body%$'\n'*}"

    if [[ "$status" -lt 200 || "$status" -ge 400 ]]; then
        (( fail_count++ ))
        printf '[fail %d/%d] HTTP %s\n' "$fail_count" "$MAX_FAILS" "$status" >&2
        (( fail_count >= MAX_FAILS )) && {
            printf '%d consecutive failures — aborting\n' "$MAX_FAILS" >&2
            exit 1
        }
        sleep "$INTERVAL"
        continue
    fi
    fail_count=0

    local current
    current=$(jq -r "$FIELD" <<< "$body" 2>/dev/null || echo "PARSE_ERROR")

    if [[ -z "$prev_value" ]]; then
        # First run — just establish baseline
        printf '[baseline] %s = %s\n' "$FIELD" "$current" >&2
    elif [[ "$current" != "$prev_value" ]]; then
        local ts; printf -v ts '%(%Y-%m-%d %H:%M:%S)T' -1
        # Log change as TSV
        printf '%s\t%s\t%s\n' "$ts" "$prev_value" "$current"

        # Optional webhook alert
        if [[ -n "$ALERT_WEBHOOK" ]]; then
            local payload
            payload=$(jq -n \
                --arg ts    "$ts"         \
                --arg url   "$URL"        \
                --arg field "$FIELD"      \
                --arg old   "$prev_value" \
                --arg new   "$current"    \
                '{ts:$ts, url:$url, field:$field, old:$old, "new":$new}')
```

```
            curl -sS -X POST \
                 -H 'Content-Type: application/json' \
                 --max-time 10 \
                 -d "$payload" "$ALERT_WEBHOOK" > /dev/null 2>&1 || true
        fi
    fi
    prev_value="$current"
    sleep "$INTERVAL"
done
```

Chapter 9 — Advanced `find` and Filesystem Operations

You already know the basics of `find` from the beginner course. This chapter digs into complex filter expressions, efficient bulk execution, parallel processing with `xargs`, and the diagnostic tools that let you see *who* has files open: `lsof`, `fuser`, and `inotifywait`.

1 — `find`: Compound Expressions

`find` builds a boolean expression tree from tests. Understanding operator precedence unlocks searches that would be impossible with a single predicate.

Operators and grouping

Operator	Meaning	Short form
`-and`	Both sides must be true (default)	(space)
`-or`	Either side true	`-o`
`-not`	Negate	!
`( … )`	Grouping — **must** be shell-escaped	

```
# Files ending in .log OR .tmp — without grouping, -or has lower precedence
find /var/app -type f \( -name '*.log' -o -name '*.tmp' \)

# NOT hidden files (name starts with .)
find . -not -name '.*'

# Owned by www-data AND world-writable
find /srv/web -type f -user www-data -perm -o+w

# Files modified in the last 7 days but not the last 1 day
find /data -type f -mtime -7 -not -mtime -1
```

Key tests quick-reference

Test	Meaning
`-name PATTERN`	Basename matches glob (case-sensitive)
`-iname PATTERN`	Same, case-insensitive
`-type f|d|l|p|s`	File / directory / symlink / FIFO / socket
`-mtime ±N`	Modified N×24 h ago (`-N`=less than, `+N`=more than)
`-mmin ±N`	Modified N minutes ago
`-newer FILE`	Modified more recently than FILE
`-size ±Nc/k/M/G`	Exact or range by size unit
`-empty`	Zero-length file or empty directory
`-perm MODE`	Exact permissions; `-MODE` = at least these bits set
`-user NAME` / `-group NAME`	Ownership
`-path PATTERN`	Full path matches glob
`-prune`	Do not descend into matched directory
`-maxdepth N`	Limit recursion depth
`-mindepth N`	Skip first N levels

Pruning subtrees for speed

```
# Skip .git directories entirely — note the -prune -o … -print pattern
find . -name '.git' -prune -o -name '*.py' -print

# Multiple directories to skip
find . \( -name '.git' -o -name '.svn' -o -name 'node_modules' \) -prune \
  -o -type f -name '*.js' -print
```

Short-circuit evaluation: `find` evaluates left to right and stops as soon as the result is determined. Put cheap tests (`-type` , `-name`) before expensive ones (`-newer` , stat-based tests) to keep things fast.

2 — Executing Commands: -exec vs -exec {} +

One-at-a-time: -exec CMD {} \;

Spawns a new process for *every* matched file. Simple, but slow for large result sets.

```
find /tmp -name '*.sock' -exec rm -f {} \;
```

Batched: -exec CMD {} +

Appends all matched paths to a single command invocation (like `xargs` built in). Much faster — one process instead of N.

```
# Delete 10 000 .tmp files — one rm call instead of 10 000
find /cache -name '*.tmp' -exec rm -f {} +

# Compress all logs older than 30 days — gzip accepts multiple files
find /var/log/app -name '*.log' -mtime +30 -exec gzip {} +
```

> **Limitation of {} +:** `{}` must be the *last* argument before `+`. You cannot write `-exec mv {} /dest/ +` — use `xargs` instead when the path needs to appear in the middle of the command.

-execdir: safer for untrusted paths

`-execdir` runs the command in the matched file's own directory, using `./filename` rather than the full path. This prevents path-traversal attacks when processing untrusted filenames.

```
# Run checksums in each directory rather than with full paths
find /uploads -type f -execdir sha256sum {} \;
```

3 — xargs in Depth

`xargs` reads items from stdin and builds command lines from them. When `-exec {} +` can't do what you need, `xargs` almost certainly can.

Core flags

Flag	Effect
`-0`	Items delimited by NUL byte — pair with `find -print0`
`-I REPLACE`	Replace string in command (default `{}`) — one item per invocation
`-n N`	Maximum N arguments per command
`-P N`	Run up to N processes in parallel
`-a FILE`	Read items from FILE instead of stdin
`-r`	Do not run command if input is empty (GNU xargs)
`-t`	Print each command before executing (trace)
`-p`	Prompt before each execution
`--max-procs=N`	Long form of `-P`

Always use `-print0 | xargs -0` for filenames

```
# Filenames with spaces break plain xargs — use NUL delimiters
find . -name '*.jpg' -print0 | xargs -0 -I{} convert {} -resize 800x {} # bad — overwrites
find . -name '*.jpg' -print0 | xargs -0 -I{} convert {} -resize 800x thumbs/{}
```

Parallel processing with `-P`

```
# Compress 8 files simultaneously — saturate CPU cores
find /archive -name '*.log' -print0 | \
  xargs -0 -P8 -n1 gzip -9

# Run a custom function in parallel — must use bash -c
process_file() {
  local f="$1"
  # … do work …
}
```

```
export -f process_file
find . -type f -print0 | xargs -0 -P4 -n1 bash -c 'process_file "$@"' _
```

Reading from a file with -a

```
# Process a pre-made list — one path per line, NUL-terminated
find /data -name '*.csv' -print0 > /tmp/csv_list.txt
# … later …
xargs -0 -a /tmp/csv_list.txt wc -l
```

Controlling batch size with -n

```
# Some commands have argument-count limits — split into batches of 100
find . -name '*.png' -print0 | xargs -0 -n100 optipng -o2
```

4 — `lsof`: List Open Files

Every open file, socket, pipe, or device in the OS can be queried with `lsof`. It answers "who has this file open?" and "what does this process have open?"

Common invocations

```
# Which process has /var/log/app.log open?
lsof /var/log/app.log

# All files open by a specific PID
lsof -p 1234

# All files open by process named nginx
lsof -c nginx

# All open files in /var/log directory
lsof +D /var/log

# All network connections (like netstat)
lsof -i
```

```
# TCP connections on port 8080
lsof -i TCP:8080

# Which process is listening on port 443?
lsof -i TCP:443 -sTCP:LISTEN

# Files opened by a specific user
lsof -u deploy
```

The deleted-but-still-open pattern

A process that holds a file open keeps the disk space allocated even after `rm` removes the name. This is the classic "disk is full but `df` and `du` disagree" scenario.

```
# Find deleted files still consuming disk space
lsof | grep 'deleted'

# Typical output — (deleted) in the NAME column:
nginx   1823  www  3w  REG  8,1  2147483648  ... /var/log/nginx/access.log (deleted)

# Fix: truncate via the /proc fd, no restart needed
> /proc/1823/fd/3
```

Useful `lsof` flags

Flag	Meaning
`-n`	No hostname lookup (faster)
`-P`	No port-name lookup (show numbers)
`-t`	Output PIDs only — scriptable
`-r N`	Repeat every N seconds
`+D DIR`	Recurse directory
`-sTCP:STATE`	Filter by TCP state (LISTEN, ESTABLISHED, ...)

```
# Kill all processes that have a specific file open
kill $(lsof -t /mnt/nas/bigfile.dat)

# Watch open connections refresh every 2 seconds
lsof -i TCP -nP -r2
```

5 — fuser: Find Processes Using Files or Sockets

`fuser` is a lighter tool than `lsof` — it focuses on a single file or socket and returns the PIDs using it. Useful for quickly answering "what is blocking this unmount?"

```
# Which processes are using this mount point?
fuser -m /mnt/usb

# Which processes are using port 80 (TCP)?
fuser 80/tcp

# Verbose — show usernames and access types
fuser -v /var/log/syslog
                     USER        PID ACCESS COMMAND
/var/log/syslog:     root        892 F....  rsyslogd

# Kill all processes using a file (be careful!)
fuser -k /var/run/app.pid

# Send a specific signal
fuser -k -HUP 80/tcp
```

Access code	Meaning
`c`	Current directory
`e`	Executable being run
`f`	Open file
`F`	Open file for writing
`r`	Root directory

`m`	mmap'd file or shared library

> **fuser vs lsof:** Use `fuser` for a quick "what's blocking unmount" check. Use `lsof` when you need the full picture — network connections, deleted files, or per-process fd lists.

6 — `inotifywait`: Watching for Filesystem Events

`inotifywait` is part of the **inotify-tools** package (Linux only). It subscribes to the kernel's inotify API, which delivers events the instant a file changes — no polling, near-zero CPU cost.

> `inotifywait` is Linux-only. On macOS use `fswatch`; on BSD use `kqueue`. The examples below assume Linux.

One-shot mode

```
# Wait until /tmp/signal.txt is created, then continue
inotifywait -e create /tmp/signal.txt
Setting up watches.
Watches established.
/tmp/ CREATE signal.txt
```

Monitor mode: `-m`

```
# Watch a directory continuously, printing all events
inotifywait -m -e create,modify,delete,moved_to /var/spool/jobs

# Parse output in a while-read loop
inotifywait -m -q --format '%e %f' -e close_write /var/spool/jobs | \
while read -r event file; do
  echo "Processing: $file ($event)"
  process_job "$file"
done
```

Recursive watching: `-r`

```
# Watch an entire directory tree
inotifywait -m -r -q --format '%w%f %e' -e modify /etc/nginx | \
while read -r path event; do
  echo "Config changed: $path"
  nginx -t && systemctl reload nginx
done
```

Common events

Event	Triggered when
`create`	File or directory created
`modify`	File contents written to
`close_write`	File written and closed — safer than `modify` for processing
`delete`	File removed
`moved_to`	File moved into watched directory
`moved_from`	File moved out of watched directory
`attrib`	Permissions, timestamps, or ownership changed
`access`	File read

Format string tokens

```
# Available tokens for --format
# %w  — directory being watched
# %f  — filename (empty if event is on the directory itself)
# %e  — event name(s), comma-separated
# %T  — timestamp (requires --timefmt)

inotifywait -m -q \
  --timefmt '%Y-%m-%d %H:%M:%S' \
  --format  '%T %e %w%f' \
  -e create,close_write \
  /var/spool/incoming
```

7 — Practical Patterns

Finding and cleaning up old files safely

```
#!/usr/bin/env bash
cleanup_old_files() {
  local dir="${1:?directory required}"
  local days="${2:-30}"
  local dry_run="${3:-false}"

  local count=0 size=0

  while IFS= read -r -d '' file; do
    fsize=$(stat -c '%s' "$file")
    (( size += fsize ))
    (( count++ ))
    if [[ $dry_run == 'false' ]]; then
      rm -f "$file"
    else
      echo "[dry-run] would delete: $file"
    fi
  done <(find "$dir" -type f -mtime +${days} -print0)

  printf 'Removed %d files (%.1f MB)\n' "$count" "$(echo "scale=1; $size/10485
76" | bc)"
}
```

Parallel checksum generation

```
# Generate SHA256 checksums for all files — 8 parallel workers
find /data/release -type f -print0 | \
  xargs -0 -P8 -n1 sha256sum >> checksums.txt

# Sort for deterministic output
sort -k2 checksums.txt -o checksums.txt
```

Auto-reload service on config change

```bash
#!/usr/bin/env bash
# watch_config.sh — reload a service whenever its config dir changes
set -euo pipefail

CONFIG_DIR="${1:?usage: $0 CONFIG_DIR SERVICE}"
SERVICE="${2:?usage: $0 CONFIG_DIR SERVICE}"

echo "Watching $CONFIG_DIR for changes…"

inotifywait -m -r -q \
  --format '%w%f' \
  -e close_write,moved_to,delete \
  "$CONFIG_DIR" | \
while IFS= read -r changed; do
  echo "Changed: $changed — reloading $SERVICE"
  if systemctl reload "$SERVICE" 2>&1; then
    echo "Reload OK"
  else
    echo "Reload FAILED — check journalctl -u $SERVICE" >&2
  fi
done
```

Find duplicate files by checksum

```bash
#!/usr/bin/env bash
find_duplicates() {
  local dir="${1:-.}"
  # Group files by size first (cheap), then checksum only same-size groups
  find "$dir" -type f -print0 | \
    xargs -0 stat --format '%s %n' | \
    sort -n | \
    awk '{ size[$1] = size[$1] " " $2; count[$1]++ }
         END { for (s in count) if (count[s]>1) print size[s] }' | \
  while read -r files; do
    # Checksum only the size-matched candidates
    # shellcheck disable=SC2086
    md5sum $files | sort | awk 'prev==$1 { print prev_f; print $2 } { prev=$1;
prev_f=$2 }'
```

```
  done
}
```

Exercises

Exercise 1 — Stale file report

Write a script `stale_report.sh DIR [DAYS]` (default 14 days) that:

- Uses `find` with NUL delimiters to list files in DIR not modified within DAYS days
- Groups them by extension and prints a summary table: extension, count, total size
- Highlights groups over 100 MB with a warning prefix

```
#!/usr/bin/env bash
set -euo pipefail

DIR="${1:?usage: $0 DIR [DAYS]}"
DAYS="${2:-14}"

declare -A ext_count ext_size

while IFS= read -r -d '' file; do
  ext="${file##*.}"
  [[ $ext == $file ]] && ext='(none)'   # no extension
  sz=$(stat -c '%s' "$file")
  (( ext_count["$ext"]++ ))
  (( ext_size["$ext"] += sz ))
done <(find "$DIR" -type f -mtime +$DAYS -print0)

printf '%-15s %8s %12s\n' 'Extension' 'Count' 'Total Size'
printf '%s\n' '----------------------------------------'

for ext in "${!ext_count[@]}"; do
  bytes="${ext_size[$ext]}"
  mb=$(( bytes / 1048576 ))
  prefix=''
  (( mb > 100 )) && prefix='[LARGE] '
```

```
  printf '%s%-15s %8d %9d MB\n' \
    "$prefix" ".$ext" "${ext_count[$ext]}" "$mb"
done | sort -k3 -rn
```

Exercise 2 — Parallel image resize

Write a script that:

- Accepts a source directory and an output directory as arguments
- Uses `find` + `xargs -P` to resize all `.jpg` and `.png` files to 800px wide in parallel (use `convert` from ImageMagick)
- Preserves directory structure under the output directory
- Skips files that already exist in the output directory

```
#!/usr/bin/env bash
set -euo pipefail

SRC="${1:?usage: $0 SRC_DIR OUT_DIR}"
OUT="${2:?usage: $0 SRC_DIR OUT_DIR}"

resize_one() {
  local src="$1" src_base="$2" out="$3"
  local rel="${src#${src_base}/}"
  local dest="$out/$rel"

  [[ -f "$dest" ]] && return 0   # already done

  mkdir -p "$(dirname "$dest")"
  convert "$src" -resize 800x "$dest"
  echo "Resized: $rel"
}

export -f resize_one
```

```
find "$SRC" \( -iname '*.jpg' -o -iname '*.png' \) -print0 | \
  xargs -0 -P"$(nproc)" -I{} bash -c 'resize_one "$@"' _ {} "$SRC" "$OUT"
```

Exercise 3 — Open-handle diagnostic

Write a function `file_users FILE` that:

- Uses `lsof` to list all processes with FILE open
- Prints a formatted table: PID, user, access mode, command name
- Returns exit code 1 if no processes have the file open (safe-to-delete signal)
- Warns if the file shows as deleted but still held open

```
file_users() {
  local file="${1:?file required}"
  local -a lines

  # -F0 outputs NUL-separated fields; easier to parse than columns
  local raw
  raw=$(lsof -nP "$file" 2>/dev/null) || true

  if [[ -z "$raw" ]]; then
    echo "No processes have $file open. Safe to delete."
    return 1
  fi

  if grep -q '(deleted)' <<<"$raw"; then
    echo "WARNING: file is deleted but still held open — disk space not r
eleased!" >&2
  fi

  printf '%-8s %-12s %-6s %s\n' 'PID' 'USER' 'MODE' 'COMMAND'
  printf '%s\n' '----------------------------------------'

  # Skip header line from lsof, parse remaining
  while read -r cmd pid user fd rest; do
    [[ $cmd == 'COMMAND' ]] && continue
    mode="${fd: -1}"     # last char of fd field: r/w/u
```

```
    printf '%-8s %-12s %-6s %s\n' "$pid" "$user" "$mode" "$cmd"
  done <<<"$raw"
}
```

Exercise 4 — File-event processor

Write a script `job_runner.sh SPOOL_DIR` that:

- Uses `inotifywait` in monitor mode to watch SPOOL_DIR for `close_write` and `moved_to` events
- On each event, processes the file (e.g. `wc -l` and move to a `done/` subdirectory)
- Logs each action with a timestamp to a logfile in the spool directory
- Handles the case where inotify-tools is not installed (print a clear error and exit 1)

```
#!/usr/bin/env bash
set -euo pipefail

SPOOL="${1:?usage: $0 SPOOL_DIR}"
LOG="$SPOOL/job_runner.log"
DONE="$SPOOL/done"

# Dependency check
if ! command -v inotifywait >/dev/null 2&1; then
  echo "ERROR: inotifywait not found. Install with: apt install inotify-t
ools" >&2
  exit 1
fi

mkdir -p "$DONE"

log() { printf '[%s] %s\n' "$(date '+%Y-%m-%d %H:%M:%S')" "$*" | tee -a
"$LOG"; }

process_job() {
  local file="$SPOOL/$1"
  [[ -f "$file" ]] || return

  local lines
```

```
  lines=$(wc -l < "$file")
  log "Processed $1: $lines lines"
  mv "$file" "$DONE/$1"
}

log "Watching $SPOOL for new jobs…"

inotifywait -m -q \
  --format '%f' \
  -e close_write,moved_to \
  "$SPOOL" | \
while IFS= read -r fname; do
  # Skip the log file and done directory itself
  [[ $fname == 'job_runner.log' || $fname == 'done' ]] && continue
  process_job "$fname"
done
```

Chapter 10 — Writing Reusable Libraries

A script that works once is useful. A library of well-designed functions that any script can source is an asset. This chapter covers how to structure Bash libraries so they are safe to source, self-documenting, version-aware, and easy to distribute — turning a collection of helper functions into something you can genuinely depend on.

1 — The Source-vs-Execute Split

Every library file needs to answer two questions: *Am I being sourced or run directly?* and *Has something already sourced me?*

The guard pattern: source-once

```
#!/usr/bin/env bash
# lib/logging.sh — safe to source multiple times

# Guard: skip re-sourcing if already loaded
[[ -n "${_LIB_LOGGING_LOADED:-}" ]] && return 0
readonly _LIB_LOGGING_LOADED=1

# … function definitions follow …
```

The `return 0` exits the sourcing operation cleanly. If the file is executed directly (no sourcing parent), `return` would fail — that's actually what you want as a hard stop for pure libraries. For libraries that also have a self-test mode, use the `BASH_SOURCE` trick instead:

```
# At the bottom of the library file
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
  # Script is being run directly — run self-tests
  _lib_logging_self_test
fi
```

> **`BASH_SOURCE[0]`** is the path of the current file. **`${0}`** is the name of the running script. They are equal only when the file is executed directly — when sourced, `${0}` remains the caller's name.

2 — Semantic Versioning Guards

When scripts depend on a specific version of a library, they should be able to assert compatibility at load time.

```
# lib/utils.sh
[[ -n "${_LIB_UTILS_LOADED:-}" ]] && return 0
readonly _LIB_UTILS_LOADED=1

readonly LIB_UTILS_VERSION='2.4.1'
readonly LIB_UTILS_VERSION_MAJOR=2
readonly LIB_UTILS_VERSION_MINOR=4

# Callers use this function to declare their minimum requirement
lib_utils_require() {
  local req_major="${1:?major version required}"
  local req_minor="${2:-0}"

  if (( LIB_UTILS_VERSION_MAJOR != req_major )); then
    printf 'ERROR: lib/utils.sh major version mismatch (have %s, need %d.x)\n'
\
      "$LIB_UTILS_VERSION" "$req_major" >&2
    return 1
  fi

  if (( LIB_UTILS_VERSION_MINOR < req_minor )); then
    printf 'ERROR: lib/utils.sh too old (have %s, need %d.%d+)\n' \
      "$LIB_UTILS_VERSION" "$req_major" "$req_minor" >&2
    return 1
  fi
}

# In a caller script:
# source lib/utils.sh
# lib_utils_require 2 3 || exit 1   # need at least 2.3
```

3 — Namespace Discipline

Bash has a single global namespace for functions and variables. Libraries must be deliberate about what they export to avoid collisions with caller scripts.

Naming conventions

Symbol type	Convention	Example
Public function	`libname_verb_noun`	`log_info`, `http_get`
Private function	`_libname_verb_noun`	`_log_format_msg`
Public constant	`LIBNAME_CONSTANT` (readonly)	`LOG_LEVEL_DEBUG`
Private variable	`_LIBNAME_VAR`	`_LOG_FD`
Guard variable	`_LIB_NAME_LOADED`	`_LIB_HTTP_LOADED`

Containing side effects with `local`

```
# Bad — _tmp leaks into the caller's scope
bad_function() {
  _tmp="$(some_command)"
  # …
}

# Good — everything stays in local scope
good_function() {
  local _tmp
  _tmp="$(some_command)"
  # …
}

# Returning values without subshells: use a nameref
str_upper() {
  local -n _result_ref="${1:?nameref var required}"
  local _input="$2"
  _result_ref="${_input^^}"   # Bash 4+ case conversion
}

# Caller:
declare upper
```

```
str_upper upper "hello world"
echo "$upper"   # HELLO WORLD — no subshell, no fork
```

> **Nameref collision hazard:** the nameref variable name and the caller's variable name must differ. If a caller passes `_result_ref` as the variable name, the nameref binds to itself. Using a double-underscore prefix (`__result`) reduces collision probability.

4 — Auto-Documentation with Structured Comments

Bash has no built-in doc system, but a consistent comment format makes it easy to generate reference docs with `grep` or `awk`.

```
##
## log_info MESSAGE [CONTEXT...]
##   Write an INFO-level log entry to stderr.
##   MESSAGE  — the primary log message
##   CONTEXT  — optional key=value pairs appended after the message
##
## Example:
##   log_info "User logged in" user=alice ip=10.0.0.1
##
log_info() {
  _log_write 'INFO' "$@"
}
```

Extracting docs at runtime

```
# Extract all ## doc blocks and their function names from a library
lib_help() {
  local libfile="${1:?library file required}"
  awk '/^##/{
    sub(/^## ?/, "")
    doc = doc $0 "\n"
    next
  }
  /^[a-zA-Z_][a-zA-Z0-9_]*\(\)/{
    if (doc != "") { print "--- " $0; printf "%s", doc; print "" }
```

```
    doc = ""
    next
  }
  { doc = "" }' "$libfile"
}
```

5 — `declare -f` Introspection

`declare -f` prints the definition of a loaded function. `declare -F` lists all defined function names. These are invaluable for debugging, mocking in tests, and building dynamic dispatch.

```bash
# List all functions defined by a library after sourcing it
before=$(declare -F)
source lib/http.sh
after=$(declare -F)
comm -13 <(sort <<<"$before") <(sort <<<"$after")   # new functions

# Print the source of a specific function
declare -f http_get

# Check if a function exists before calling it
func_exists() { declare -f "$1" >/dev/null 2&1; }

if func_exists http_retry; then
  http_retry "$url"
else
  http_get "$url"
fi
```

Dynamic dispatch via function name construction

```bash
# A plugin pattern — call handler_TYPE if it exists, else default
dispatch() {
  local type="$1"; shift
  local handler="handle_${type}"

  if func_exists "$handler"; then
```

```
    "$handler" "$@"
  else
    printf 'No handler for type: %s\n' "$type" >&2
    return 1
  fi
}

handle_json() { echo "Handling JSON: $*"; }
handle_csv()  { echo "Handling CSV: $*"; }

dispatch json data.json   # → handle_json data.json
dispatch xml  data.xml    # → No handler for type: xml
```

6 — A Complete Library Template

Here is a production-ready skeleton you can copy as a starting point for any new library:

```
#!/usr/bin/env bash
# ============================================================================
=
# lib/logging.sh — structured log helpers
# Version: 1.0.0  |  Requires: bash >= 4.2
# ============================================================================
=

# Source guard
[[ -n "${_LIB_LOGGING_LOADED:-}" ]] && return 0
readonly _LIB_LOGGING_LOADED=1

# Bash version check
if (( BASH_VERSINFO[0] < 4 || ( BASH_VERSINFO[0] == 4 && BASH_VERSINFO[1] < 2
) )); then
  printf 'ERROR: lib/logging.sh requires bash >= 4.2 (have %s)\n' \
    "$BASH_VERSION" >&2
  return 1
fi

# --- Public constants ------------------------------------------------------
-
```

```
readonly LIB_LOGGING_VERSION='1.0.0'
readonly LOG_LEVEL_DEBUG=0
readonly LOG_LEVEL_INFO=1
readonly LOG_LEVEL_WARN=2
readonly LOG_LEVEL_ERROR=3

# --- Private state -----------------------------------------------------------
-
_LOG_LEVEL="${LOG_LEVEL:-$LOG_LEVEL_INFO}"   # caller may set LOG_LEVEL
_LOG_FILE="${LOG_FILE:-}"                    # empty = stderr only
_LOG_COLOR="${LOG_COLOR:-auto}"              # auto | always | never

# Resolve colour mode once at load time
if [[ $_LOG_COLOR == 'auto' ]]; then
  [[ -t 2 ]] && _LOG_COLOR='always' || _LOG_COLOR='never'
fi

# ANSI colours
if [[ $_LOG_COLOR == 'always' ]]; then
  _C_DEBUG='\e[0;36m'   # cyan
  _C_INFO='\e[0;32m'    # green
  _C_WARN='\e[0;33m'    # yellow
  _C_ERROR='\e[0;31m'   # red
  _C_RESET='\e[0m'
else
  _C_DEBUG='' _C_INFO='' _C_WARN='' _C_ERROR='' _C_RESET=''
fi

# --- Private helpers ---------------------------------------------------------
-
_log_write() {
  local level_name="$1"; shift
  local level_num="$1"; shift
  (( level_num < _LOG_LEVEL )) && return 0

  local color_var="_C_${level_name}"
  local ts="$(date '+%Y-%m-%dT%H:%M:%S')"
  local line
  printf -v line '%s[%s] [%s] %s%s' \
    "${!color_var}" "$ts" "$level_name" "$*" "$_C_RESET"
```

```
  printf '%s\n' "$line" >&2
  [[ -n "$_LOG_FILE" ]] && printf '%s\n' "$line" >> "$_LOG_FILE"
}

# --- Public API ----------------------------------------------------------
-
##
## log_debug MESSAGE
##   Write a DEBUG entry (visible when LOG_LEVEL=0)
##
log_debug() { _log_write DEBUG "$LOG_LEVEL_DEBUG" "$@"; }

##
## log_info MESSAGE
##
log_info()  { _log_write INFO  "$LOG_LEVEL_INFO"  "$@"; }

##
## log_warn MESSAGE
##
log_warn()  { _log_write WARN  "$LOG_LEVEL_WARN"  "$@"; }

##
## log_error MESSAGE
##   Write an ERROR entry and return exit code 1
##
log_error() { _log_write ERROR "$LOG_LEVEL_ERROR" "$@"; return 1; }

##
## log_set_level LEVEL
##   Set minimum log level: 0=DEBUG 1=INFO 2=WARN 3=ERROR
##
log_set_level() { _LOG_LEVEL="${1:?level required}"; }

##
## log_set_file PATH
##   Tee all log output to PATH in addition to stderr
##
log_set_file() { _LOG_FILE="${1:?path required}"; }

# --- Self-test -----------------------------------------------------------
```

```
-
_lib_logging_self_test() {
  echo "=== lib/logging.sh self-test ==="
  log_debug "This is DEBUG"
  log_info  "This is INFO"
  log_warn  "This is WARN"
  log_error "This is ERROR" || true
  log_set_level "$LOG_LEVEL_WARN"
  log_info  "You should NOT see this"
  log_warn  "You should see this"
  echo "=== PASS ==="
}

if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
  _lib_logging_self_test
fi
```

7 — Distribution and Sourcing Strategies

Portable sourcing: finding the library relative to the script

```
#!/usr/bin/env bash
# Works whether the script is on PATH, symlinked, or called by absolute path
SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
source "$SCRIPT_DIR/../lib/logging.sh"
source "$SCRIPT_DIR/../lib/utils.sh"
```

> `pwd -P` resolves symlinks. This ensures the path is absolute and canonical even when the script is invoked via a symlink in `/usr/local/bin`.

A loader function for larger projects

```
# lib/loader.sh — source multiple libraries with error checking
_LIB_ROOT=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)

lib_load() {
```

```
  local lib
  for lib in "$@"; do
    local path="$_LIB_ROOT/${lib}.sh"
    if [[ ! -f "$path" ]]; then
      printf 'lib_load: library not found: %s\n' "$path" >&2
      return 1
    fi
    # shellcheck source=/dev/null
    source "$path" || { printf 'lib_load: failed to source %s\n' "$path" >&2;
return 1; }
  done
}

# Usage in a script:
# source /opt/myapp/lib/loader.sh
# lib_load logging utils http   # sources logging.sh, utils.sh, http.sh
```

Single-file distribution via bundling

```
# bundle.sh — concatenate all library files into one distributable
# Strip duplicate shebangs and source guards, keeping functions
{
  echo '#!/usr/bin/env bash'
  echo '# Generated bundle — do not edit manually'
  for lib in lib/*.sh; do
    echo; echo "# ── $lib ──"
    # Remove shebang lines from non-first files
    grep -v '^#!' "$lib"
  done
} > dist/myapp_bundle.sh
chmod +x dist/myapp_bundle.sh
```

8 — Testing Your Library

A library without tests is a library you'll break silently. Even a minimal inline self-test loop is better than nothing.

```bash
# Minimal test runner — no external dependencies
_TESTS_RUN=0
_TESTS_FAILED=0

assert_eq() {
  local desc="$1" expected="$2" actual="$3"
  (( _TESTS_RUN++ ))
  if [[ "$expected" == "$actual" ]]; then
    printf '  PASS: %s\n' "$desc"
  else
    printf '  FAIL: %s\n    expected: %q\n    actual:   %q\n' \
      "$desc" "$expected" "$actual"
    (( _TESTS_FAILED++ ))
  fi
}

assert_exit() {
  local desc="$1" expected_code="$2"
  shift 2
  (( _TESTS_RUN++ ))
  "$@"; local rc=$?
  if (( rc == expected_code )); then
    printf '  PASS: %s (exit %d)\n' "$desc" "$rc"
  else
    printf '  FAIL: %s (expected exit %d, got %d)\n' \
      "$desc" "$expected_code" "$rc"
    (( _TESTS_FAILED++ ))
  fi
}

test_summary() {
  printf '\nResults: %d/%d passed\n' \
    "$(( _TESTS_RUN - _TESTS_FAILED ))" "$_TESTS_RUN"
  (( _TESTS_FAILED == 0 ))   # exit 0 on success
}

# Example usage
assert_eq "str_upper converts hello" "HELLO" "$(str_upper <<< 'hello')"
assert_exit "log_error returns 1" 1 log_error "test error"
test_summary
```

Exercises

Exercise 1 — lib/validate.sh

Write a library `lib/validate.sh` with the following public functions, each returning exit 0 on success and printing an error message on failure:

- `validate_nonempty VAR_NAME VALUE` — fails if VALUE is empty
- `validate_integer VAR_NAME VALUE` — fails if VALUE is not a whole number
- `validate_file VAR_NAME PATH` — fails if PATH does not exist as a readable file
- `validate_dir VAR_NAME PATH` — fails if PATH is not a directory

Include source guard, version constant, and a self-test that runs when executed directly.

```
#!/usr/bin/env bash
# lib/validate.sh — input validation helpers  v1.0.0

[[ -n "${_LIB_VALIDATE_LOADED:-}" ]] && return 0
readonly _LIB_VALIDATE_LOADED=1
readonly LIB_VALIDATE_VERSION='1.0.0'

_validate_fail() {
  printf 'validate: %s: %s\n' "$1" "$2" >&2
  return 1
}

##
## validate_nonempty VAR_NAME VALUE
##
validate_nonempty() {
  [[ -n "${2:-}" ]] || _validate_fail "$1" "must not be empty"
}

##
## validate_integer VAR_NAME VALUE
##
validate_integer() {
  [[ "${2:-}" =~ ^[0-9]+$ ]] || \
    _validate_fail "$1" "'${2:-}' is not a non-negative integer"
```

```
}

##
## validate_file VAR_NAME PATH
##
validate_file() {
  [[ -f "${2:-}" && -r "${2:-}" ]] || \
    _validate_fail "$1" "'${2:-}' is not a readable file"
}

##
## validate_dir VAR_NAME PATH
##
validate_dir() {
  [[ -d "${2:-}" ]] || _validate_fail "$1" "'${2:-}' is not a directory"
}

_lib_validate_self_test() {
  local pass=0 fail=0
  _chk() {
    if "$@" 2>/dev/null; then (( pass++ )); else (( fail++ )); fi
  }
  _nochk() {
    if ! "$@" 2>/dev/null; then (( pass++ )); else (( fail++ )); fi
  }
  _chk   validate_nonempty X "hello"
  _nochk validate_nonempty X ""
  _chk   validate_integer  N "42"
  _nochk validate_integer  N "abc"
  _chk   validate_file     F /etc/hostname
  _nochk validate_file     F /no/such/file
  _chk   validate_dir      D /tmp
  _nochk validate_dir      D /etc/hostname
  printf 'validate self-test: %d pass, %d fail\n' "$pass" "$fail"
  (( fail == 0 ))
}

[[ "${BASH_SOURCE[0]}" == "${0}" ]] && _lib_validate_self_test
```

Exercise 2 — Nameref return values

Write a library `lib/strings.sh` using nameref (`local -n`) for all return values — no subshells. Include:

- `str_trim RETVAR INPUT` — strip leading and trailing whitespace
- `str_repeat RETVAR STR N` — repeat STR N times
- `str_join RETVAR SEP ARRAY_NAME` — join array elements with SEP

```bash
#!/usr/bin/env bash
# lib/strings.sh  v1.0.0

[[ -n "${_LIB_STRINGS_LOADED:-}" ]] && return 0
readonly _LIB_STRINGS_LOADED=1

##
## str_trim RETVAR INPUT
##
str_trim() {
  local -n __str_trim_ret="$1"
  local __s="$2"
  __s="${__s#"${__s%%[![:space:]]*}"}"   # strip leading
  __s="${__s%"${__s##*[![:space:]]}"}"   # strip trailing
  __str_trim_ret="$__s"
}

##
## str_repeat RETVAR STR N
##
str_repeat() {
  local -n __str_repeat_ret="$1"
  local __str="$2" __n="$3" __acc=''
  local __i
  for (( __i=0; __i<__n; __i++ )); do
    __acc+="$__str"
  done
  __str_repeat_ret="$__acc"
}
```

```
##
## str_join RETVAR SEP ARRAY_NAME
##   ARRAY_NAME is the name of an existing array variable
##
str_join() {
  local -n __str_join_ret="$1"
  local __sep="$2"
  local -n __arr="$3"
  local __out='' __elem
  for __elem in "${__arr[@]}"; do
    [[ -n $__out ]] && __out+="$__sep"
    __out+="$__elem"
  done
  __str_join_ret="$__out"
}

[[ "${BASH_SOURCE[0]}" == "${0}" ]] && {
  declare result
  str_trim result "  hello world  "
  echo "trim:   '$result'"
  str_repeat result "ab" 4
  echo "repeat: '$result'"
  words=( one two three )
  str_join result ', ' words
  echo "join:   '$result'"
}
```

Exercise 3 — Auto-doc extractor

Write a script `libdoc.sh LIBFILE` that:

- Parses the `##` doc-comment blocks from a library file (as shown in section 4)
- Prints a clean reference page with each function's signature and description
- Also accepts `--list` to print just the function names, one per line
- Exits 1 if no documented functions are found

```bash
#!/usr/bin/env bash
set -euo pipefail

MODE='full'
if [[ "${1:-}" == '--list' ]]; then
  MODE='list'; shift
fi
LIBFILE="${1:?usage: $0 [--list] LIBFILE}"
[[ -f $LIBFILE ]] || { echo "Not found: $LIBFILE" >&2; exit 1; }

count=0

awk -v mode="$MODE" '
BEGIN { doc = ""; in_doc = 0 }
/^##/ {
  sub(/^## ?/, "")
  doc = doc $0 "\n"
  in_doc = 1
  next
}
in_doc && /^[a-zA-Z_][a-zA-Z0-9_]*[[:space:]]*\(\)/ {
  match($0, /^[a-zA-Z_][a-zA-Z0-9_]*/)
  fname = substr($0, RSTART, RLENGTH)
  if (mode == "list") {
    print fname
  } else {
    print "\033[0;36m" fname "\033[0m"
    printf "%s", doc
    print ""
  }
  count++
  doc = ""; in_doc = 0
  next
}
{ doc = ""; in_doc = 0 }
END { exit (count == 0 ? 1 : 0) }
' "$LIBFILE"
```

Exercise 4 — Loader with dependency resolution

Extend the loader pattern so libraries can declare their own dependencies. Write `lib/loader.sh` where each library file may include a line like:

```
# REQUIRES: utils logging
```

The loader should parse this line and automatically source the listed libraries first, detecting circular dependencies and aborting with a clear error if one is found.

```
#!/usr/bin/env bash
# lib/loader.sh — dependency-aware library loader

[[ -n "${_LIB_LOADER_LOADED:-}" ]] && return 0
readonly _LIB_LOADER_LOADED=1

_LIB_ROOT=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
declare -gA _LIB_LOADING   # currently in-flight (cycle detection)

lib_load() {
  local lib
  for lib in "$@"; do
    local guard="_LIB_${lib^^}_LOADED"
    [[ -n "${!guard:-}" ]] && continue   # already loaded

    if [[ -n "${_LIB_LOADING[$lib]:-}" ]]; then
      printf 'lib_load: circular dependency detected: %s\n' "$lib" >&2
      return 1
    fi

    local path="$_LIB_ROOT/$lib.sh"
    [[ -f $path ]] || { printf 'lib_load: not found: %s\n' "$path" >&2; r
eturn 1; }

    # Parse REQUIRES line (e.g.  # REQUIRES: utils logging)
    local req_line
    req_line=$(grep -m1 '^# REQUIRES:' "$path" || true)
    if [[ -n $req_line ]]; then
      local -a deps
```

```
        # shellcheck disable=SC2206
        deps=( ${req_line#*: } )
        _LIB_LOADING[$lib]=1
        lib_load "${deps[@]}" || return 1
        unset '_LIB_LOADING[$lib]'
      fi

      # shellcheck source=/dev/null
      _LIB_LOADING[$lib]=1
      source "$path" || return 1
      unset '_LIB_LOADING[$lib]'
    done
}
```

Chapter 11 — Building Complete CLI Tools

There is a gap between "a script that works for me" and "a tool I can hand to a colleague." Closing that gap means robust option parsing, a helpful `--help` page, a `--version` flag, shell tab completion, and — for complex tools — a man-page stub. This chapter builds all of it.

1 — Long-Option Parsing with `getopt`

The shell built-in `getopts` handles short options only. For long options (`--verbose`, `--output FILE`) you need the external `getopt` (note: no *s*), which reorders and normalises the argument list before your loop sees it.

> **Two different tools:** `getopts` (built-in, POSIX, short options only) vs `getopt` (external GNU utility, long options, requires `--` sentinel). Always use the GNU version: `getopt --version` should say "util-linux". The BSD version shipped on macOS is limited — install GNU getopt via Homebrew if needed.

```
#!/usr/bin/env bash
set -euo pipefail

PROG="$(basename "$0")"
VERSION='1.2.0'

# Define accepted options
#   Short: -v -o FILE -q -h
#   Long:  --verbose --output FILE --quiet --help --version
_OPTS=$(getopt \
  --name    "$PROG" \
  --options 'vo:qh' \
  --longoptions 'verbose,output:,quiet,help,version' \
  -- "$@") || { usage; exit 1; }

# Replace positional parameters with the normalised list
eval set -- "$_OPTS"
```

```
# Defaults
VERBOSE=0
QUIET=0
OUTPUT=''

while true; do
  case "$1" in
    -v | --verbose)  VERBOSE=1;     shift   ;;
    -q | --quiet)    QUIET=1;       shift   ;;
    -o | --output)   OUTPUT="$2";  shift 2 ;;
    -h | --help)     usage;         exit 0  ;;
         --version)  version;       exit 0  ;;
    --)              shift;         break   ;;
    *)               usage;         exit 1  ;;
  esac
done

# Remaining positional arguments
ARGS=( "$@" )
```

After `eval set -- "$_OPTS"`, `getopt` guarantees that option arguments are separate tokens, abbreviated options are expanded, and `--` marks the end of options. Your `while` loop just needs to handle the canonical forms.

A purely manual alternative (no external tool)

When you cannot depend on GNU `getopt` being present, a manual `while/case` loop handles most real-world needs:

```
while [[ $# -gt 0 ]]; do
  case "$1" in
    -v | --verbose)  VERBOSE=1; shift ;;
    -o | --output)
      [[ $# -lt 2 ]] && { echo "$1 requires an argument" >&2; exit 1; }
      OUTPUT="$2"; shift 2 ;;
    --output=*)  OUTPUT="${1#--output=}"; shift ;;   # --output=FILE form
    -h | --help)    usage; exit 0 ;;
    --version)      version; exit 0 ;;
    --)             shift; break ;;
    -*)             echo "Unknown option: $1" >&2; exit 1 ;;
    *)              break ;;   # first non-option: stop parsing
```

```
    esac
  done
```

2 — Generating `--help` Output

A good help page answers: what does the tool do, how do I call it, what are my options, and where can I find more. Keep it concise but complete.

Heredoc approach — simple and readable

```
usage() {
  cat <<EOF
Usage: $PROG [OPTIONS] FILE [FILE...]

Process one or more input FILEs and write results to stdout or --output.

Options:
  -o, --output FILE    Write output to FILE instead of stdout
  -v, --verbose        Show detailed progress
  -q, --quiet          Suppress all non-error output
  -h, --help           Show this help and exit
      --version        Show version and exit

Examples:
  $PROG input.txt
  $PROG --output result.txt *.log
  $PROG --verbose --output /dev/stderr data.csv

Exit codes:
  0  Success
  1  Usage error
  2  Input file not found
  3  Permission denied
EOF
}
```

Usage-string extraction — DRY alternative

Embed the usage string in a comment block at the top of the script, then extract it at runtime. This keeps the help in one place and makes it visible to `grep` without running the tool:

```
#!/usr/bin/env bash
# USAGE: mytool [OPTIONS] FILE [FILE...]
#
# Process input files.
#
# Options:
#   -o, --output FILE   Output destination
#   -v, --verbose       Verbose output
#   -h, --help          Show this help
#
PROG="$(basename "$0")"

usage() {
  # Print every line between '# USAGE:' and the first non-comment line
  sed -n '/^# USAGE:/,/^[^#]/{ /^[^#]/d; s/^# \?//; p }' "${BASH_SOURCE[0]}"
}
```

3 — `--version` and Exit Codes

```
version() {
  printf '%s %s\n' "$PROG" "$VERSION"
}

# Conventional exit codes (document them in --help)
readonly EX_OK=0          # success
readonly EX_USAGE=1       # bad arguments / options
readonly EX_NOINPUT=2     # input file missing
readonly EX_NOPERM=3      # permission denied
readonly EX_SOFTWARE=4    # internal error

# Consistent die() helper — always writes to stderr
die() {
  local code="${1}"; shift
  printf '%s: %s\n' "$PROG" "$*" >&2
  exit "$code"
```

```
}

# Usage:
# die "$EX_NOINPUT" "file not found: $file"
# die "$EX_USAGE"   "--output requires an argument"
```

> Distinguish between **usage errors** (wrong arguments — print a usage hint and exit 1) and **runtime errors** (file not found, network timeout — print the error and exit with a specific code). Users and scripts calling your tool depend on predictable exit codes.

4 — Subcommand Dispatch

Tools that do more than one thing (like `git` or `docker`) use subcommands. The cleanest pattern keeps each subcommand in its own function and dispatches by name:

```
#!/usr/bin/env bash
# myapp — a tool with subcommands
set -euo pipefail
PROG="$(basename "$0")"

cmd_init()   { echo "Initialising..."; }
cmd_build()  { echo "Building...";     }
cmd_deploy() { echo "Deploying...";    }

cmd_help() {
  cat <<EOF
Usage: $PROG COMMAND [OPTIONS]

Commands:
  init     Initialise a new project
  build    Build the project
  deploy   Deploy to the configured target
  help     Show this help
EOF
}

# Dispatch — fail fast with a helpful message for unknown commands
COMMAND="${1:-help}"; shift || true
```

```
if declare -f "cmd_${COMMAND}" >/dev/null 2&1; then
  "cmd_${COMMAND}" "$@"
else
  printf '%s: unknown command: %s\n\n' "$PROG" "$COMMAND" >&2
  cmd_help >&2
  exit 1
fi
```

5 — Shell Tab Completion with `bash_completion`

Tab completion transforms a tool from something you look up into something you *discover*. The bash completion system calls a function you register via `complete` whenever the user presses Tab after your tool's name.

How it works

```
# The completion system sets these variables before calling your function:
#   COMP_WORDS  — array of all words on the current command line
#   COMP_CWORD  — index of the word currently being completed
#   COMPREPLY   — array your function fills with completion candidates

_mytool_complete() {
  local cur="${COMP_WORDS[$COMP_CWORD]}"
  local prev="${COMP_WORDS[$COMP_CWORD-1]}"

  # Complete option arguments
  case "$prev" in
    -o | --output)
      # Complete filenames
      COMPREPLY=( $(compgen -f -- "$cur") )
      return ;;
    --format)
      COMPREPLY=( $(compgen -W 'json csv tsv text' -- "$cur") )
      return ;;
  esac

  # Complete options and subcommands when the user typed a dash or nothing
  if [[ $cur == -* ]]; then
```

```
    COMPREPLY=( $(compgen -W \
      '--verbose --quiet --output --format --help --version' \
      -- "$cur") )
  else
    COMPREPLY=( $(compgen -W 'init build deploy help' -- "$cur") )
  fi
}

# Register the completion function for mytool
complete -F _mytool_complete mytool
```

Key compgen actions

Flag	Completes
`-W 'word1 word2'`	Fixed word list
`-f`	Filenames in current directory
`-d`	Directories only
`-u`	Usernames
`-g`	Group names
`-c`	Commands on PATH
`-A function`	Function names
`-v`	Shell variable names

Distributing the completion file

```
# Users source this from ~/.bashrc, or you install it system-wide:
#   /etc/bash_completion.d/mytool       — system-wide (Debian/Ubuntu)
#   /usr/share/bash-completion/completions/mytool  — XDG location
#   ~/.local/share/bash-completion/completions/mytool  — per-user

# The file itself contains only the function definition and the complete call.
# mytool.bash-completion:
_mytool_complete() { # … as above … }
complete -F _mytool_complete mytool
```

Dynamic completions: instead of a hard-coded `-W` word list, call your own tool to get completions at runtime. Many tools support a hidden `--complete` flag that prints valid values for a given context. This keeps the completion script and the tool in sync automatically.

6 — Dynamic Completions from the Tool Itself

```
# --complete-subcommands and --complete-options flags for self-describing tool
s
case "${1:-}" in
  --complete-subcommands)
    # Print all subcommand names, one per line
    declare -F | awk '/^declare -f cmd_/{ sub(/^declare -f cmd_/, ""); print
}'
    exit 0 ;;
  --complete-options)
    printf '%s\n' '--verbose' '--quiet' '--output' '--format' '--help' '--vers
ion'
    exit 0 ;;
esac

# Completion script uses the tool itself to stay in sync
_mytool_complete() {
  local cur="${COMP_WORDS[$COMP_CWORD]}"
  local prev="${COMP_WORDS[$COMP_CWORD-1]}"

  if [[ $cur == -* ]]; then
    COMPREPLY=( $(compgen -W "$(mytool --complete-options)" -- "$cur") )
  else
    COMPREPLY=( $(compgen -W "$(mytool --complete-subcommands)" -- "$cur") )
  fi
}
complete -F _mytool_complete mytool
```

7 — Man-Page Stubs

A man page is the canonical reference. Writing full nroff/troff is painful; `help2man` generates a reasonable first draft from your `--help` and `--version` output. For a hand-crafted stub that covers most use cases:

```
#!/bin/sh
# Generate and preview a man page stub
# Install help2man: apt install help2man / brew install help2man

help2man --no-info --name 'process input files' \
  --output mytool.1 ./mytool

# Preview
man -l mytool.1

# Install
sudo cp mytool.1 /usr/local/share/man/man1/
sudo mandb
```

Minimal hand-written man stub format

```
.\" mytool.1 — minimal man page
.TH MYTOOL 1 "$(date '+%B %Y')"" "1.2.0" "User Commands"
.SH NAME
mytool \- process input files
.SH SYNOPSIS
.B mytool
[\fIOPTIONS\fR] \fIFILE\fR [\fIFILE\fR...]
.SH DESCRIPTION
Process one or more input files and write results to stdout.
.SH OPTIONS
.TP
\fB\-o\fR, \fB\-\-output\fR \fIFILE\fR
Write output to FILE instead of stdout.
.TP
\fB\-v\fR, \fB\-\-verbose\fR
Show detailed progress messages.
.TP
\fB\-h\fR, \fB\-\-help\fR
Display help and exit.
```

```
.SH EXIT STATUS
.TP
.B 0
Success.
.TP
.B 1
Usage error.
.SH AUTHOR
Your Name <you@example.com>
```

8 — Putting It All Together: A Reference Template

Here is a complete, production-ready CLI tool skeleton combining everything above:

```bash
#!/usr/bin/env bash
# ============================================================================
=
# mytool — one-line description
#
# USAGE: mytool [OPTIONS] FILE [FILE...]
#
# Options:
#   -o, --output FILE   Write output to FILE (default: stdout)
#   -v, --verbose       Increase verbosity
#   -q, --quiet         Suppress non-error output
#   -h, --help          Show this help and exit
#       --version       Show version and exit
# ============================================================================
=
set -euo pipefail

PROG="$(basename "${BASH_SOURCE[0]}")"
VERSION='1.0.0'

# ── Exit codes ──────────────────────────────────────────────────────────────
──
readonly EX_OK=0 EX_USAGE=1 EX_NOINPUT=2 EX_NOPERM=3

# ── Helpers ─────────────────────────────────────────────────────────────────
```

```
──
die()     { printf '%s: %s\n' "$PROG" "$*" >&2; exit "${_exit_code:-1}"; }
warn()    { (( QUIET )) || printf '%s: warning: %s\n' "$PROG" "$*" >&2; }
verbose() { (( VERBOSE )) && printf '%s\n' "$*" >&2 || true; }

usage() {
  sed -n '/^# USAGE:/,/^# ===/{/^# ===/d; s/^# \?//; p}' \
    "${BASH_SOURCE[0]}"
}
version() { printf '%s %s\n' "$PROG" "$VERSION"; }

# ── Argument parsing ──────────────────────────────────────────────
──
VERBOSE=0
QUIET=0
OUTPUT=''

while [[ $# -gt 0 ]]; do
  case "$1" in
    -v|--verbose)  VERBOSE=1;          shift   ;;
    -q|--quiet)    QUIET=1;            shift   ;;
    -o|--output)   OUTPUT="${2:?--output requires FILE}"; shift 2 ;;
    --output=*)   OUTPUT="${1#--output=}"; shift   ;;
    -h|--help)     usage;              exit 0  ;;
    --version)     version;            exit 0  ;;
    --)            shift;             break   ;;
    -*)            _exit_code=$EX_USAGE die "unknown option: $1" ;;
    *)             break                       ;;
  esac
done

(( $# > 0 )) || { usage >&2; exit "$EX_USAGE"; }

# ── Output setup ──────────────────────────────────────────────────
──
if [[ -n $OUTPUT ]]; then
  exec 1> "$OUTPUT"   # redirect stdout to file for the rest of the script
fi

# ── Main logic ────────────────────────────────────────────────────
──
```

```bash
process_file() {
  local f="$1"
  [[ -f "$f" ]] || { _exit_code=$EX_NOINPUT die "file not found: $f"; }
  [[ -r "$f" ]] || { _exit_code=$EX_NOPERM  die "permission denied: $f"; }
  verbose "Processing: $f"
  # … actual work …
  cat "$f"
}

for file in "$@"; do
  process_file "$file"
done
```

Exercises

Exercise 1 — Robust option parser

Write a script `csvcut.sh` that accepts the following interface:

```
Usage: csvcut [OPTIONS] FILE [FILE...]

  -f, --fields LIST     Comma-separated field numbers or names (required)
  -d, --delimiter CHAR  Input delimiter (default: ,)
  -H, --no-header       Input has no header row
  -o, --output FILE     Write to FILE instead of stdout
  -h, --help
      --version
```

Use a manual `while/case` loop. Validate that `--fields` was provided; exit 1 with a usage hint if it was not. Actual CSV processing can be a stub (`echo "Would process: $*"`) — focus on the option handling.

```bash
#!/usr/bin/env bash
# USAGE: csvcut [OPTIONS] FILE [FILE...]
#
#   -f, --fields LIST     Comma-separated field numbers (required)
```

```
#   -d, --delimiter CHAR  Input delimiter (default: ,)
#   -H, --no-header       Input has no header row
#   -o, --output FILE     Write to FILE instead of stdout
#   -h, --help            Show this help
#       --version         Show version
#
set -euo pipefail
PROG="$(basename "$0")"
VERSION='1.0.0'

usage()   { sed -n '/^# USAGE:/,/^#$/{ s/^# \?//; p }' "${BASH_SOURCE
[0]}"; }
version() { printf '%s %s\n' "$PROG" "$VERSION"; }
die()     { printf '%s: %s\n' "$PROG" "$*" >&2; exit 1; }

FIELDS=''
DELIM=','
NO_HEADER=0
OUTPUT=''

while [[ $# -gt 0 ]]; do
  case "$1" in
    -f|--fields)     FIELDS="${2:?--fields requires LIST}";    shift 2 ;;
    --fields=*)      FIELDS="${1#--fields=}";                   shift
;;
    -d|--delimiter)  DELIM="${2:?--delimiter requires CHAR}";   shift 2
;;
    --delimiter=*)   DELIM="${1#--delimiter=}";                 shift
;;
    -H|--no-header)  NO_HEADER=1;                               shift
;;
    -o|--output)     OUTPUT="${2:?--output requires FILE}";     shift 2
;;
    --output=*)      OUTPUT="${1#--output=}";                   shift
;;
    -h|--help)       usage; exit 0                              ;;
    --version)       version; exit 0                            ;;
    --)              shift; break                               ;;
    -*)              die "unknown option: $1"                   ;;
    *)               break                                      ;;
  esac
```

```
done

[[ -n $FIELDS ]] || { usage >&2; die "--fields is required"; }
(( $# > 0 )) || { usage >&2; die "at least one FILE is required"; }

[[ -n $OUTPUT ]] && exec 1>"$OUTPUT"

printf 'fields=%s delim=%q no_header=%d files=%s\n' \
  "$FIELDS" "$DELIM" "$NO_HEADER" "$*"
# Real CSV processing would go here
```

Exercise 2 — Subcommand tool

Build a tool `vaultctl` with subcommands `add`, `get`, `list`, and `delete` that manages a simple key=value store in `~/.vaultctl/store`. Requirements:

- Each subcommand has its own `--help`
- The top-level `help` command lists all subcommands with one-line descriptions
- Unknown subcommands print an error and list the available ones
- `add KEY VALUE`, `get KEY`, `list`, `delete KEY`

```
#!/usr/bin/env bash
set -euo pipefail
PROG="$(basename "$0")"
STORE_DIR="${HOME}/.vaultctl"
STORE="$STORE_DIR/store"
mkdir -p "$STORE_DIR"
chmod 700 "$STORE_DIR"
: "${STORE:?}"   # ensure path is non-empty

cmd_add() {
  [[ "${1:-}" == '--help' ]] && { echo "Usage: $PROG add KEY VALUE"; retu
rn; }
  local key="${1:?add requires KEY}" val="${2:?add requires VALUE}"
  # Remove existing key then append
  sed -i "/^${key}=/d" "$STORE" 2>/dev/null || true
  printf '%s=%s\n' "$key" "$val" >> "$STORE"
```

```
  chmod 600 "$STORE"
  echo "Stored: $key"
}

cmd_get() {
  [[ "${1:-}" == '--help' ]] && { echo "Usage: $PROG get KEY"; return; }
  local key="${1:?get requires KEY}"
  local val
  val=$(grep -m1 "^${key}=" "$STORE" 2>/dev/null || true)
  [[ -n $val ]] || { echo "key not found: $key" >&2; return 1; }
  printf '%s\n' "${val#*=}"
}

cmd_list() {
  [[ "${1:-}" == '--help' ]] && { echo "Usage: $PROG list"; return; }
  [[ -f $STORE ]] || { echo "(empty)"; return; }
  awk -F= '{ printf "%-20s %s\n", $1, $2 }' "$STORE"
}

cmd_delete() {
  [[ "${1:-}" == '--help' ]] && { echo "Usage: $PROG delete KEY"; return;
}
  local key="${1:?delete requires KEY}"
  sed -i "/^${key}=/d" "$STORE" && echo "Deleted: $key"
}

cmd_help() {
  cat <<EOF
Usage: $PROG COMMAND [ARGS]

Commands:
  add    KEY VALUE   Store a key/value pair
  get    KEY         Retrieve a value by key
  list               List all stored keys
  delete KEY         Remove a key
  help               Show this help

Run '$PROG COMMAND --help' for command-specific help.
EOF
}
```

```
CMD="${1:-help}"; shift || true
if declare -f "cmd_${CMD}" >/dev/null 2&1; then
  "cmd_${CMD}" "$@"
else
  printf '%s: unknown command: %s\n\n' "$PROG" "$CMD" >&2
  cmd_help >&2
  exit 1
fi
```

Exercise 3 — Tab completion script

Write a bash completion file `vaultctl.bash-completion` for the `vaultctl` tool from Exercise 2. It should:

- Complete subcommand names at the first position
- Complete `--help` at the second position for any subcommand
- For `get` and `delete`, complete available key names by reading `~/.vaultctl/store`

```
# vaultctl.bash-completion
# Source in ~/.bashrc or install to /etc/bash_completion.d/

_vaultctl_complete() {
  local cur="${COMP_WORDS[$COMP_CWORD]}"
  local prev="${COMP_WORDS[$COMP_CWORD-1]}"
  local subcmd="${COMP_WORDS[1]:-}"
  local store="${HOME}/.vaultctl/store"

  # Position 1: complete subcommands
  if (( COMP_CWORD == 1 )); then
    COMPREPLY=( $(compgen -W 'add get list delete help' -- "$cur") )
    return
  fi

  # Position 2+: subcommand-specific completions
  case "$subcmd" in
    get|delete)
      # Complete key names from store file
      local keys=''
```

```
      [[ -f $store ]] && keys=$(awk -F= '{ print $1 }' "$store")
      COMPREPLY=( $(compgen -W "$keys --help" -- "$cur") ) ;;
    add)
      # After 'add KEY', nothing to complete (VALUE is free-form)
      (( COMP_CWORD == 2 )) &&
        COMPREPLY=( $(compgen -W '--help' -- "$cur") ) ;;
    list|help)
      COMPREPLY=( $(compgen -W '--help' -- "$cur") ) ;;
  esac
}

complete -F _vaultctl_complete vaultctl
```

Exercise 4 — Self-contained tool skeleton

Using the reference template from section 8, create a complete tool `lsrecent` that:

- Lists files modified within the last N hours (default: 24)
- Accepts `-n/--hours N`, `-s/--sort (name|size|time)`, `-l/--long` (show size and timestamp), `-h/--help`, `--version`
- Takes an optional directory argument (default: current directory)
- Includes an embedded usage-string comment and a completion-aware `--complete-options` hidden flag

---

```
#!/usr/bin/env bash
# USAGE: lsrecent [OPTIONS] [DIR]
#
# List files modified within the last N hours.
#
# Options:
#   -n, --hours N       Lookback window in hours (default: 24)
#   -s, --sort FIELD    Sort by: name, size, time (default: time)
#   -l, --long          Show size and modification timestamp
#   -h, --help          Show this help and exit
#       --version       Show version and exit
#
set -euo pipefail
```

```
PROG="$(basename "${BASH_SOURCE[0]}")"
VERSION='1.0.0'

usage()   { sed -n '/^# USAGE:/,/^#$/{ s/^# \?//; p }' "${BASH_SOURCE
[0]}"; }
version() { printf '%s %s\n' "$PROG" "$VERSION"; }
die()     { printf '%s: %s\n' "$PROG" "$*" >&2; exit 1; }

# Hidden flag for completion
[[ "${1:-}" == '--complete-options' ]] && {
  printf '%s\n' '--hours' '--sort' '--long' '--help' '--version'; exit 0;
}

HOURS=24
SORT='time'
LONG=0

while [[ $# -gt 0 ]]; do
  case "$1" in
    -n|--hours)    HOURS="${2:?--hours requires N}";    shift 2 ;;
    --hours=*)    HOURS="${1#--hours=}";               shift   ;;
    -s|--sort)     SORT="${2:?--sort requires FIELD}";  shift 2 ;;
    --sort=*)      SORT="${1#--sort=}";                 shift   ;;
    -l|--long)     LONG=1;                              shift   ;;
    -h|--help)     usage; exit 0                        ;;
    --version)     version; exit 0                      ;;
    --)            shift; break                          ;;
    -*)            die "unknown option: $1"              ;;
    *)             break                                 ;;
  esac
done

DIR="${1:-.}"
[[ -d $DIR ]] || die "not a directory: $DIR"
[[ $SORT =~ ^(name|size|time)$ ]] || die "--sort must be name, size, or t
ime"
[[ $HOURS =~ ^[0-9]+$ ]] || die "--hours must be a positive integer"

# find uses minutes; convert hours
MINS=$(( HOURS * 60 ))
```

```
if (( LONG )); then
  # Print: size mtime path
  find "$DIR" -type f -mmin -${MINS} -print0 | \
    xargs -0 stat --format '%s %y %n' | \
    case $SORT in
      name) sort -k3  ;;
      size) sort -k1 -rn ;;
      time) sort -k2 -r  ;;
    esac
else
  case $SORT in
    name) find "$DIR" -type f -mmin -${MINS} | sort ;;
    size) find "$DIR" -type f -mmin -${MINS} -print0 |
            xargs -0 stat --format '%s %n' | sort -rn | awk '{ print $2
}' ;;
    time) find "$DIR" -type f -mmin -${MINS} -print0 |
            xargs -0 stat --format '%Y %n' | sort -rn | awk '{ print $2
}' ;;
  esac
fi
```

Chapter 12 — Scripting for Cron and Systemd

A script that works perfectly when you run it interactively can silently fail when a scheduler runs it at 3 AM. The reasons are almost always environmental: missing PATH entries, no terminal, no home directory, or a second copy launched before the first one finished. This chapter covers everything you need to write scripts that behave correctly when run unattended.

1 — Cron Syntax and the Crontab

A crontab line has five time fields followed by the command:

```
# ┌──────────── minute        (0–59)
# │  ┌──────────── hour          (0–23)
# │  │  ┌──────────── day of month  (1–31)
# │  │  │  ┌──────────── month         (1–12 or JAN–DEC)
# │  │  │  │  ┌──────────── day of week   (0–7, 0 and 7 = Sunday, or SUN–SAT)
# │  │  │  │  │
# *  *  *  *  *  command

# Run at 02:30 every day
30  2  *  *  *  /opt/myapp/bin/backup.sh

# Every 15 minutes
*/15  *  *  *  *  /opt/myapp/bin/poll.sh

# 9 AM on weekdays only
0  9  *  *  1-5  /opt/myapp/bin/report.sh

# First day of every month at midnight
0  0  1  *  *  /opt/myapp/bin/monthly.sh

# Multiple values with commas
0  8,12,17  *  *  *  /opt/myapp/bin/check.sh
```

Managing crontabs

```
# Edit your crontab interactively
crontab -e

# List current crontab
crontab -l

# Install from a file (replaces entire crontab)
crontab crontab.txt

# Non-interactively add a line — safe even if crontab is empty
( crontab -l 2>/dev/null; echo "30 2 * * * /opt/myapp/bin/backup.sh" ) | cront
ab -

# Remove a specific line (match and delete)
crontab -l | grep -v 'backup.sh' | crontab -

# Edit another user's crontab (as root)
crontab -u deploy -e

# System-wide crontab — has an extra USER field
# /etc/crontab and files in /etc/cron.d/:
# 30 2 * * *  deploy  /opt/myapp/bin/backup.sh
```

> **Tip:** use crontab.guru to verify your schedule expressions interactively before deploying them.

2 — The Cron Environment Problem

Cron runs jobs with a minimal environment — typically just `HOME`, `LOGNAME`, and a very short `PATH` (often just `/usr/bin:/bin`). Everything your interactive shell loads — `~/.bashrc`, `nvm`, `pyenv`, `/usr/local/bin` — is absent.

The three defences

```
# 1. Set PATH explicitly at the top of your crontab
PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

```
MAILTO=ops@example.com    # send output/errors here; empty string = discard
SHELL=/bin/bash

30 2 * * * backup.sh
```

```
# 2. Use absolute paths for every binary inside the script
#!/usr/bin/env bash
PYTHON=/usr/bin/python3
RSYNC=/usr/bin/rsync
CURL=/usr/bin/curl
```

```
# 3. Source your profile at the top of the script (when you NEED its env)
#!/usr/bin/env bash
# Deliberately load the user environment — use sparingly
# shellcheck source=/dev/null
[[ -f /etc/profile ]] && source /etc/profile
[[ -f "${HOME}/.bashrc" ]] && source "${HOME}/.bashrc"
```

Diagnosing environment issues

```
# Simulate the cron environment locally
env -i HOME="$HOME" LOGNAME="$LOGNAME" PATH='/usr/bin:/bin' \
  SHELL=/bin/bash bash --norc --noprofile ./myscript.sh

# Dump the actual cron environment to a file for inspection
* * * * * env > /tmp/cron-env.txt 2&1
```

3 — Locking: Preventing Overlapping Runs

Without a lock, a slow job can still be running when cron starts the next instance. Two copies writing to the same file or database is rarely safe.

flock — the right tool for the job

```
# Inline — entire script protected in one line from the crontab
* * * * *  flock -n /var/lock/myjob.lock /opt/myapp/bin/myjob.sh

# -n = non-blocking: exit immediately if lock is held (skip this run)
# -w N = wait up to N seconds before giving up
flock -w 10 /var/lock/myjob.lock /opt/myapp/bin/myjob.sh
```

flock inside the script

```
#!/usr/bin/env bash
set -euo pipefail

LOCKFILE=/var/lock/myjob.lock

# Open the lock file on fd 9 and attempt an exclusive non-blocking lock
exec 9>"$LOCKFILE"
if ! flock -n 9; then
  echo "Another instance is already running. Exiting." >&2
  exit 1
fi
# Lock is held for the rest of the script; released automatically on exit

echo "Running job..."
# … work …
```

> **Why not PID files?** Classic PID-file locking is race-prone and leaves stale files after crashes. `flock` uses kernel-level advisory locks that are automatically released when the process exits — no cleanup needed.

Logging with timestamps for unattended jobs

```
#!/usr/bin/env bash
LOGFILE=/var/log/myapp/myjob.log
MAXLOG=10485760   # 10 MB — rotate when exceeded

log() { printf '[%s] %s\n' "$(date '+%Y-%m-%dT%H:%M:%S')" "$*"; }
```

```
# Rotate log if too large
if [[ -f $LOGFILE ]] && (( $(stat -c '%s' "$LOGFILE") > MAXLOG )); then
  mv "$LOGFILE" "$LOGFILE.$(date +%Y%m%d%H%M%S)"
fi

# Redirect all output (stdout + stderr) to the log file
exec >>"$LOGFILE" 2&1

log "=== Job started (PID $$) ==="
# … work …
log "=== Job finished ==="
```

4 — Systemd Timer Units

Systemd timers are the modern alternative to cron. They offer better logging (via journald), dependency management, missed-run handling, and per-service resource controls.

Every timer needs two files: a **.service** unit (what to run) and a **.timer** unit (when to run it).

The service unit

```
# /etc/systemd/system/myjob.service
[Unit]
Description=My scheduled job
After=network-online.target
Wants=network-online.target

[Service]
Type=oneshot
User=deploy
Group=deploy
WorkingDirectory=/opt/myapp
ExecStart=/opt/myapp/bin/myjob.sh
StandardOutput=journal
StandardError=journal
SyslogIdentifier=myjob

# Resource limits — prevent runaway jobs
TimeoutStartSec=300
```

```
MemoryMax=512M
CPUQuota=50%

[Install]
WantedBy=multi-user.target
```

The timer unit

```
# /etc/systemd/system/myjob.timer
[Unit]
Description=Run myjob daily at 02:30

[Timer]
# Calendar expression — like cron but human-readable
OnCalendar=*-*-* 02:30:00

# If the system was off at the scheduled time, run ASAP on next boot
Persistent=true

# Spread load — randomise start within 5 minutes of the scheduled time
RandomizedDelaySec=5min

[Install]
WantedBy=timers.target
```

Timer management commands

```
# Enable and start the timer
sudo systemctl daemon-reload
sudo systemctl enable --now myjob.timer

# Check timer status and next run time
systemctl status myjob.timer
systemctl list-timers --all

# Run the job immediately (without waiting for the timer)
sudo systemctl start myjob.service
```

```
# View recent logs
journalctl -u myjob.service -n 50

# Follow logs in real time
journalctl -u myjob.service -f

# Logs since last run
journalctl -u myjob.service --since "$(systemctl show myjob.service -p ExecMai
nStartTimestamp --value)"
```

OnCalendar expressions

Expression	Meaning
`daily`	Every day at midnight (shorthand)
`hourly`	Every hour at :00
`weekly`	Every Monday at midnight
`monthly`	First day of every month at midnight
`*-*-* 02:30:00`	Every day at 02:30
`Mon..Fri *-*-* 09:00:00`	Weekdays at 09:00
`*-*-1 00:00:00`	First of every month at midnight
`*:0/15`	Every 15 minutes

```
# Test a calendar expression without running anything
systemd-analyze calendar 'Mon..Fri *-*-* 09:00:00'
  Original form: Mon..Fri *-*-* 09:00:00
Normalized form: Mon..Fri *-*-* 09:00:00
    Next elapse: Mon 2026-06-09 09:00:00 UTC
       (in UTC): Mon 2026-06-09 09:00:00 UTC
```

5 — User Timers (No Root Required)

You don't need root to use systemd timers. User timers live under `~/.config/systemd/user/` and run as your own user.

```
# ~/.config/systemd/user/sync.service
[Unit]
Description=Sync notes to remote

[Service]
Type=oneshot
ExecStart=%h/bin/sync-notes.sh   # %h expands to $HOME

# ~/.config/systemd/user/sync.timer
[Unit]
Description=Sync notes every 30 minutes

[Timer]
OnCalendar=*:0/30
Persistent=true

[Install]
WantedBy=timers.target
```

```
# Manage user timers — no sudo needed
systemctl --user daemon-reload
systemctl --user enable --now sync.timer
systemctl --user list-timers
journalctl --user -u sync.service -f

# Allow user timers to run even when logged out
sudo loginctl enable-linger "$USER"
```

6 — A Production-Ready Cron Script Template

This template handles every common unattended-job concern in one place:

```
#!/usr/bin/env bash
# ================================================================================
=
# myjob.sh — nightly data export
# Safe for cron and systemd. Run as: deploy user.
```

```
# ================================================================================
=
set -euo pipefail

# ── Identity ──────────────────────────────────────────────────────────────────
──
PROG="$(basename "${BASH_SOURCE[0]}")"
PIDSTR="[$$]"

# ── Environment ───────────────────────────────────────────────────────────────
──
PATH=/usr/local/bin:/usr/bin:/bin

# ── Paths ─────────────────────────────────────────────────────────────────────
──
APP_DIR=/opt/myapp
LOG_DIR=/var/log/myapp
LOCK_FILE=/var/lock/myjob.lock
LOG_FILE="$LOG_DIR/myjob.log"
mkdir -p "$LOG_DIR"

# ── Logging ───────────────────────────────────────────────────────────────────
──
exec >>"$LOG_FILE" 2&1
log()  { printf '[%s] %s %s\n'    "$(date '+%Y-%m-%dT%H:%M:%S')" "$PIDSTR"
"$*"; }
warn() { printf '[%s] %s WARN %s\n' "$(date '+%Y-%m-%dT%H:%M:%S')" "$PIDSTR"
"$*"; }

# ── Locking ───────────────────────────────────────────────────────────────────
──
exec 9>"$LOCK_FILE"
if ! flock -n 9; then
  log "Already running — skipping this invocation"
  exit 0
fi

# ── Traps ─────────────────────────────────────────────────────────────────────
──
on_exit() {
  local rc=$?
```

```
  if (( rc != 0 )); then
    log "=== Job FAILED (exit $rc) ==="
  else
    log "=== Job finished OK ==="
  fi
}
trap on_exit EXIT

# ── Main ──────────────────────────────────────────────────────────
log "=== Job started ==="

# … your work here …
log "Exporting data..."
# export_data ...

log "Done."
```

7 — Sending Alerts on Failure

Silent failures are the worst kind. Build notification into the job itself, not just the crontab's `MAILTO` setting (which requires a working MTA).

```
#!/usr/bin/env bash
SLACK_WEBHOOK="${SLACK_WEBHOOK_URL:?set SLACK_WEBHOOK_URL}"
PROG="$(basename "$0")"
HOST=$(hostname -s)

notify_failure() {
  local exit_code="$1"
  local message
  printf -v message '*FAILED* `%s` on `%s` (exit %d) at %s' \
    "$PROG" "$HOST" "$exit_code" "$(date '+%Y-%m-%d %H:%M:%S')"
  curl -s -X POST -H 'Content-Type: application/json' \
    --data "{\"text\":\"${message}\"}" \
    "$SLACK_WEBHOOK" || true   # never let alerting kill the job
}

on_exit() {
```

```
  local rc=$?
  (( rc != 0 )) && notify_failure "$rc"
}
trap on_exit EXIT

# … rest of script …
```

Using systemd OnFailure for alert routing

```
# /etc/systemd/system/myjob.service
[Unit]
Description=My job
OnFailure=notify-failure@%n.service   # %n = unit name

# /etc/systemd/system/notify-failure@.service  (template unit)
[Unit]
Description=Notify on failure for %i

[Service]
Type=oneshot
ExecStart=/opt/myapp/bin/notify-failure.sh %i
```

8 — Cron vs Systemd Timers: Quick Comparison

Feature	Cron	Systemd timer
Availability	Universal	Linux with systemd only
Logging	Email / redirect manually	journald — automatic, queryable
Missed runs	Silently skipped	`Persistent=true` catches up
Dependencies	None	After=, Wants=, Requires=
Resource limits	None built-in	Memory, CPU, IO per unit
Overlapping runs	Requires manual locking	Default: one instance at a time
User jobs (no root)	`crontab -e`	`~/.config/systemd/user/`

Schedule syntax	5-field numeric	Human-readable calendar strings
Randomised jitter	Manual with sleep $RANDOM	`RandomizedDelaySec=`

Exercises

Exercise 1 — Hardened cron script

Write `db_backup.sh` — a script meant to run daily from cron that:

- Sets an explicit `PATH` at the top
- Uses `flock` on fd 9 to prevent overlapping runs
- Redirects all output to `/var/log/myapp/db_backup.log` with timestamps
- Rotates the log file when it exceeds 5 MB
- Traps EXIT to log success or failure
- The actual "backup" can be a stub (`sleep 2 && echo "Backup complete"`)

```
#!/usr/bin/env bash
set -euo pipefail

PATH=/usr/local/bin:/usr/bin:/bin

LOG_DIR=/var/log/myapp
LOG_FILE="$LOG_DIR/db_backup.log"
LOCK_FILE=/var/lock/db_backup.lock
MAX_LOG=5242880   # 5 MB

mkdir -p "$LOG_DIR"

# Rotate log if needed
if [[ -f $LOG_FILE ]] && \
   (( $(stat -c '%s' "$LOG_FILE") > MAX_LOG )); then
  mv "$LOG_FILE" "$LOG_FILE.$(date +%Y%m%d%H%M%S)"
fi

exec >>"$LOG_FILE" 2&1
```

```
log() { printf '[%s] [%d] %s\n' "$(date '+%Y-%m-%dT%H:%M:%S')" "$$" "$*"; }

# Lock
exec 9>"$LOCK_FILE"
if ! flock -n 9; then
  log "Already running — skipping"
  exit 0
fi

on_exit() {
  local rc=$?
  if (( rc == 0 )); then
    log "=== FINISHED OK ==="
  else
    log "=== FAILED (exit $rc) ==="
  fi
}
trap on_exit EXIT

log "=== DB backup started ==="
# Stub backup work
sleep 2
log "Backup complete"
```

Exercise 2 — Systemd timer unit

Write a pair of systemd unit files for a weekly report generator:

- `weekly-report.service` — runs `/opt/reports/generate.sh` as user `reports`, depends on network being available, journals all output, times out after 10 minutes
- `weekly-report.timer` — fires every Monday at 06:00, catches up on missed runs, spreads load with up to 10 minutes of random jitter

Also write the three `systemctl` commands needed to deploy and start the timer without rebooting.

```
# /etc/systemd/system/weekly-report.service
[Unit]
```

```
Description=Weekly report generator
After=network-online.target
Wants=network-online.target

[Service]
Type=oneshot
User=reports
Group=reports
ExecStart=/opt/reports/generate.sh
StandardOutput=journal
StandardError=journal
SyslogIdentifier=weekly-report
TimeoutStartSec=600

[Install]
WantedBy=multi-user.target
```

```
# /etc/systemd/system/weekly-report.timer
[Unit]
Description=Run weekly report every Monday at 06:00

[Timer]
OnCalendar=Mon *-*-* 06:00:00
Persistent=true
RandomizedDelaySec=10min

[Install]
WantedBy=timers.target
```

```
# Deploy commands
sudo systemctl daemon-reload
sudo systemctl enable weekly-report.timer
sudo systemctl start  weekly-report.timer
```

Exercise 3 — crontab management script

Write a script `cron-manage.sh` with subcommands:

- `add SCHEDULE COMMAND` — add an entry if it doesn't already exist (match on COMMAND to avoid duplicates)
- `remove PATTERN` — remove all entries whose command matches PATTERN
- `list` — pretty-print the current crontab, skipping comment lines
- `check` — verify every command path in the crontab exists and is executable

---

```
#!/usr/bin/env bash
set -euo pipefail
PROG="$(basename "$0")"

cmd_add() {
  local schedule="${1:?add requires SCHEDULE}"
  local command="${2:?add requires COMMAND}"
  local entry="$schedule $command"
  local current
  current=$(crontab -l 2>/dev/null || true)

  if grep -qF "$command" <<<"$current" 2>/dev/null; then
    echo "Entry already exists for: $command"
    return 0
  fi

  ( echo "$current"; echo "$entry" ) | grep -v '^$' | crontab -
  echo "Added: $entry"
}

cmd_remove() {
  local pattern="${1:?remove requires PATTERN}"
  crontab -l 2>/dev/null | grep -v "$pattern" | crontab -
  echo "Removed entries matching: $pattern"
}

cmd_list() {
  crontab -l 2>/dev/null | grep -v '^[[:space:]]*#' | \
  grep -v '^[[:space:]]*$' | \
  awk '{ printf "%-40s %s\n", $1" "$2" "$3" "$4" "$5, substr($0, index
($0,$6)) }'
}
```

```
cmd_check() {
  local ok=0 bad=0
  while read -r m h dom mon dow cmd rest...; do
    # cmd is the first token after the 5 time fields
    if [[ -x "$cmd" ]]; then
      printf '  OK:      %s\n' "$cmd"; (( ok++ ))
    else
      printf '  MISSING: %s\n' "$cmd"; (( bad++ ))
    fi
  done <(crontab -l 2>/dev/null | grep -v '^[[:space:]]*[#$]')
  printf '%d OK, %d missing\n' "$ok" "$bad"
  (( bad == 0 ))
}

CMD="${1:-list}"; shift || true
if declare -f "cmd_${CMD}" >/dev/null 2&1; then
  "cmd_${CMD}" "$@"
else
  printf '%s: unknown command: %s\n' "$PROG" "$CMD" >&2
  exit 1
fi
```

Exercise 4 — Resilient job wrapper

Write a general-purpose wrapper script `run-job.sh COMMAND [ARGS...]` that can be used from both cron and systemd to wrap any command with:

- Exclusive locking based on a hash of the command path (so two different jobs don't share a lock)
- Timestamped logging to `/var/log/jobs/JOBNAME.log` where JOBNAME is the basename of the wrapped command
- A timeout: `TIMEOUT` env var in seconds (default 3600) after which the job is killed
- Exit code pass-through — the wrapper exits with the same code as the wrapped command
- A summary line: `FINISHED|FAILED|TIMEOUT — duration Xs`

```
#!/usr/bin/env bash
set -euo pipefail
```

```
CMD="${1:?usage: $0 COMMAND [ARGS...]}"
JOBNAME="$(basename "$CMD")"
TIMEOUT="${TIMEOUT:-3600}"
LOG_DIR=/var/log/jobs
LOG_FILE="$LOG_DIR/$JOBNAME.log"

# Lock file based on the full command path (md5 to keep filename safe)
LOCK_HASH=$(printf '%s' "$CMD" | md5sum | cut -c1-8)
LOCK_FILE="/var/lock/job_${LOCK_HASH}.lock"

mkdir -p "$LOG_DIR"
exec >>"$LOG_FILE" 2&1

ts()  { date '+%Y-%m-%dT%H:%M:%S'; }
log() { printf '[%s] %s\n' "$(ts)" "$*"; }

exec 9>"$LOCK_FILE"
if ! flock -n 9; then
  log "LOCKED — $JOBNAME already running, skipping"
  exit 0
fi

START=$(date '+%s')
log "START $JOBNAME (timeout=${TIMEOUT}s, pid=$$)"

exit_code=0
timed_out=0

# Run with timeout — timeout exits 124 on expiry
timeout --kill-after=5 "$TIMEOUT" "$@" || {
  exit_code=$?
  (( exit_code == 124 )) && timed_out=1
}

ELAPSED=$(( $(date '+%s') - START ))

if (( timed_out )); then
  log "TIMEOUT — $JOBNAME killed after ${ELAPSED}s"
elif (( exit_code != 0 )); then
  log "FAILED — $JOBNAME exit $exit_code after ${ELAPSED}s"
else
```

```
  log "FINISHED — $JOBNAME OK in ${ELAPSED}s"
fi

exit "$exit_code"
```