Bash Scripting — Advanced Course

12-chapter advanced-level course

CONTENTS

1. Advanced I/O Redirection & File Descriptors
2. Subshells, Environments & Scope
3. Robust Error Handling & Exit Codes
4. Bash Internals: Parsing, Expansion & Execution
5. Advanced Process Management
6. Writing Portable & POSIX-compatible Scripts
7. Building DSLs in Bash
8. Advanced Testing with BATS
9. Daemon Scripts & Process Management
10. Dynamic Dispatch & Metaprogramming
11. The Bash Completion System
12. Integrating Bash with Other Languages

Chapter 1 — Bash Internals: Parsing and Expansion Order

Most Bash bugs are not logic errors — they are expansion order surprises. When a line of shell script behaves unexpectedly, the cause is almost always that the programmer had a different mental model of *when* the shell interprets each piece of syntax than what actually happens. This chapter builds that model precisely, from the moment Bash reads a character to the moment a process is launched.

1 — Tokenisation: What the Lexer Sees

Before any expansion happens, Bash reads its input and breaks it into **tokens**. A token is either a *word* (a sequence of characters treated as a unit) or an *operator* (a metacharacter like `|`, `;>`, or `()`).

The rules the lexer applies — in strict priority order:

1. **Quoting.** Any character preceded by `\`, or any sequence inside `'…'` or `"…"`, is shielded from further lexer processing. Quoting is resolved *during* tokenisation, not during expansion.
2. **Operator recognition.** If the current character starts or continues an operator token (`|`, `&`, `;`, `(`, `)`, `<`, `>`, and their two-character forms), it becomes an operator token — not a word.
3. **Whitespace delimiters.** Unquoted `IFS` characters (space, tab, newline by default) delimit words but are not themselves tokens.
4. **Comments.** An unquoted `#` that appears as the first character of a word starts a comment; everything to the end of the line is discarded before parsing.

```
# Illustrating tokenisation — what are the tokens here?
echo hello | cat
# Tokens: [echo] [hello] [|] [cat]
# 'hello' and 'cat' are WORDS; '|' is an OPERATOR

# Quoting prevents operator recognition
echo 'hello | cat'
# Tokens: [echo] [hello | cat]
# The pipe is INSIDE a quoted word — not an operator

# The # rule: only a word-initial # is a comment
```

```
echo foo#bar   # prints: foo#bar  — # is not word-initial here
echo foo #bar  # prints: foo      — space before # makes it word-initial
```

2 — The Seven-Stage Expansion Pipeline

After tokenisation, each *word* token is passed through up to seven expansions, applied strictly left-to-right in this order. Understanding the order is everything — a result from one stage feeds directly into the next.

#	Expansion	Trigger syntax
1	Brace expansion	`{a,b} {1..5}`
2	Tilde expansion	`~ ~user ~+ ~-`
3	Parameter & variable expansion	`$var ${var} ${!var}`
4	Arithmetic expansion	`$(( expr ))`
5	Command substitution	`$( cmd ) `cmd``
6	Word splitting	(implicit — splits on `$IFS`)
7	Pathname expansion (globbing)	`* ? […]`

> **Process substitution** (`<(cmd)` / `>(cmd)`) is handled concurrently with command substitution but is not part of the standard seven-stage pipeline — it is a Bash extension that produces a filename (`/dev/fd/N`) before the command line is assembled.

Stage 1 — Brace expansion

Brace expansion happens *first*, before any variable is looked up. This has a critical implication: you cannot put a variable inside braces to control expansion.

```
# Works as expected
echo file{1,2,3}.txt
file1.txt file2.txt file3.txt

# Sequence expressions
echo {a..e}
```

```
a b c d e
echo {01..05}
01 02 03 04 05
echo {0..20..5}    # with step (bash 4+)
0 5 10 15 20

# Brace expansion is purely textual — no variable lookup
list='a,b,c'
echo {$list}         # does NOT expand list — prints: {a,b,c}
eval echo {$list}    # eval forces re-parsing: prints a b c

# Preamble and postscript are repeated for each element
mkdir -p project/{src,test,docs}/{main,util}
# creates: project/src/main, project/src/util, project/test/main, ...
```

Stage 2 — Tilde expansion

```
# ~ expands to $HOME
echo ~          # /home/alice
echo ~root      # /root  (another user's home)
echo ~+         # $PWD (current directory)
echo ~-         # $OLDPWD (previous directory)

# Tilde expansion ONLY fires when ~ is the first char of a word
# and the word is unquoted
echo "~"        # prints literal ~  (quoted)
echo /foo/~     # prints /foo/~    (not word-initial)

# Assignment tilde expansion fires after = and each :
PATH=~/bin:$PATH   # ~ expands here inside an assignment
```

Stage 3 — Parameter expansion

This is the richest stage. The key advanced points:

```
# Indirect expansion — ${!var} looks up the value OF the variable named by var
target='PATH'
echo "${!target}"   # prints the value of $PATH
```

```
# ${!prefix*} and ${!prefix@} — variable names matching a prefix
FOO_A=1; FOO_B=2
echo "${!FOO_*}"    # FOO_A FOO_B

# Substring extraction: ${var:offset:length}
s='hello world'
echo "${s:6}"       # world
echo "${s:6:3}"     # wor
echo "${s: -5}"     # world  (negative offset — note the space!)

# Pattern removal
f='/path/to/file.tar.gz'
echo "${f##*/}"     # file.tar.gz   (greedy from left)
echo "${f%.*}"      # /path/to/file.tar  (lazy from right)
echo "${f%%.*}"     # /path/to/file  (greedy from right)

# Case conversion (bash 4+)
x='Hello World'
echo "${x,,}"       # hello world
echo "${x^^}"       # HELLO WORLD
echo "${x^}"        # Hello World (first char only)
```

Stages 4 & 5 — Arithmetic and command substitution

Both produce a string that re-enters the pipeline at stage 6 (word splitting). They are evaluated in the order they appear left-to-right within a word, but each is fully resolved before the result is passed forward.

```
# Arithmetic expansion — integer math only, no fork
n=5
echo $(( n * 3 + 1 ))     # 16
echo $(( 0xFF ))           # 255  (hex literal)
echo $(( 2**10 ))          # 1024  (exponentiation)

# Command substitution — the output feeds back as a word
today=$(date '+%Y-%m-%d')

# CRITICAL: trailing newlines are stripped by command substitution
x=$(printf 'foo\n\n\n')
echo "'${x}'"   # 'foo'  — all trailing newlines gone
```

```
# Preserve trailing newlines with a sentinel character
x=$(printf 'foo\n\n'; echo x)
x="${x%x}"          # strip the sentinel, keep the newlines
```

Stage 6 — Word splitting

This stage is the source of the majority of quoting bugs. It applies *only* to the results of parameter expansion, arithmetic expansion, and command substitution — **never** to literal text already in the script.

```
# IFS controls word splitting — default is space, tab, newline
words='one two   three'
set -- $words        # word-split: $1=one $2=two $3=three
set -- "$words"      # quoted: $1='one two   three'

# Changing IFS changes what counts as a delimiter
IFS=':'
set -- $PATH         # splits PATH into positional parameters
IFS=$' \t\n'         # restore default

# IFS='' disables word splitting entirely — useful in while-read loops
while IFS= read -r line; do
  # $line is NEVER split, even if it contains spaces or tabs
  echo "$line"
done < file.txt

# Word splitting does NOT apply inside [[ … ]] or (( … ))
x='a b'
[[ $x == 'a b' ]] && echo "match"   # safe — [[ never word-splits
# [ $x == 'a b' ] would fail — old-style [ word-splits
```

Stage 7 — Pathname expansion (globbing)

```
# Globbing fires after word splitting — on each resulting word
pattern='*.txt'
ls $pattern      # splits then globs: ls file1.txt file2.txt
ls "$pattern"    # quoted: ls literally looks for a file named '*.txt'
```

```bash
# Extended globs (shopt -s extglob)
shopt -s extglob
ls !(*.log)      # everything except .log files
ls +(foo|bar)*   # one or more of 'foo' or 'bar', then anything
ls @(*.jpg|*.png)  # exactly one of the alternatives

# globstar for recursive matching (bash 4+)
shopt -s globstar
ls **/*.py        # all .py files in any subdirectory

# If no match: by default, the pattern is passed literally
# With nullglob: no-match expands to nothing (empty)
shopt -s nullglob
files=( *.xyz )
(( "${#files[@]}" == 0 )) && echo "no matches"

# failglob: treat no-match as an error instead
shopt -s failglob
```

3 — Where Quoting Lives in This Model

A persistent confusion: people think quoting is about expansion. It is not. Quoting is resolved by the *lexer*, before any expansion stage runs. What quoting actually does is mark characters as *not subject to the rules that apply to unquoted characters in later stages*.

Quote form	What it prevents	What still happens inside it
`'…'`	Everything — no expansion at all	Nothing
`"…"`	Word splitting, globbing, brace expansion, tilde expansion	Parameter, arithmetic, command substitution; `\` before `$ ` " \ newline`
`\x`	All special meaning of the next character	Nothing (the backslash is consumed)
`$'…'`	All expansion; enables C-style escapes	`\n \t \xNN \uNNNN` etc.

```
# Double-quote prevents word splitting and globbing but NOT substitution
f='hello world'
echo "The value is: $f"      # parameter expansion fires inside ""
echo "Result: $((2+2))"       # arithmetic expansion fires inside ""
echo "Files: $(ls)"           # command substitution fires inside ""
echo "Glob: *.txt"            # glob does NOT fire inside "" — literal

# $'...' for control characters without $() or printf
NL=$'\n'
TAB=$'\t'
ESC=$'\e'   # or \033
printf 'col1%scol2\n' "$TAB"
```

4 — Evaluation Contexts That Change the Rules

The seven stages apply to simple commands. Several other constructs have different evaluation rules that catch experienced programmers off guard.

[[…]] — the compound test command

```
# Inside [[: word splitting and pathname expansion are DISABLED
# Pattern matching on the right-hand side of == is still active

x='foo bar'
[[ $x == 'foo bar' ]]       # safe without quotes — no splitting
[[ $x == foo* ]]             # RHS is a PATTERN (not a string)
[[ $x == 'foo*' ]]           # quoted RHS = literal string match
[[ $x =~ ^foo[[:space:]] ]]  # RHS is an ERE regex — never quote it

# BASH_REMATCH captures regex groups
if [[ '2026-06-09' =~ ^([0-9]{4})-([0-9]{2})-([0-9]{2})$ ]]; then
  echo "year=${BASH_REMATCH[1]} month=${BASH_REMATCH[2]} day=${BASH_REMATCH
[3]}"
fi
```

((…)) — arithmetic evaluation context

```
# Inside (()): variable names do NOT need $
# String values that look like integers are evaluated as integers
# An expression that evaluates to 0 returns exit code 1 (false)

a=5; b=3
(( a > b )) && echo "a is larger"
(( a += b ))           # a is now 8 — side effects are allowed

# 0 is false, non-zero is true — opposite of shell exit codes
(( 0 )) || echo "zero is false"
(( 1 )) && echo "non-zero is true"

# Pitfall: (( expression )) with -e set — false exits the script!
set -e
count=0
(( count++ ))   # DANGER: count++ evaluates to 0 (the old value), exits script
# Safe pattern:
(( count++ )) || true
# or:
count=$(( count + 1 ))    # $(()) always succeeds
```

Here-strings and here-documents

```
# Here-string: a single word, expanded then fed as stdin
# Expansion stages 3-5 apply; no word splitting or globbing
cat <<<"Hello $USER, you have $(ls | wc -l) files"

# Heredoc: delimiter controls expansion
# Unquoted delimiter: stages 3-5 apply
cat << EOF
Hello $USER
EOF

# Quoted delimiter ('EOF', "EOF", \EOF): NO expansion — literal text
cat << 'EOF'
Hello $USER   <-- this is literal, not expanded
EOF
```

5 — Alias Expansion: Stage Zero

Aliases are not part of the seven-stage pipeline. They are expanded by the *parser*, before any other processing — including before brace expansion. This makes them powerful but also fragile.

```
# Aliases are expanded during parsing, not execution
# They apply only to interactive shells and scripts that explicitly enable the
m
shopt -s expand_aliases   # required in non-interactive scripts

alias ll='ls -la'
ll /tmp          # alias expanded before ll is parsed as a command

# Aliases vs functions: aliases cannot take arguments by position
# Functions are nearly always preferable in scripts
# The ONE valid use case for aliases in scripts: redefining a command globally
alias grep='grep --color=auto'
```

6 — Practical Consequences: Reading Bugs Through the Model

Let's apply the model to diagnose real bugs you will encounter.

Bug: glob fires when you don't want it to

```
pattern='*.log'
if [[ -n $pattern ]]; then
  # Safe: [[ never globs
fi

# Dangerous: passes to an external command unquoted
find . -name $pattern   # if *.log matches files HERE, it expands before find
sees it
find . -name "$pattern" # correct — quote prevents premature glob
```

Bug: array element count broken by word splitting

```
# Wrong: word splitting turns one element into many
items='one two three'
arr=( $items )       # three elements — probably what you want
arr=( "$items" )     # ONE element: 'one two three' — probably NOT what you wa
nt

# Use mapfile for line-delimited data instead
mapfile -t arr < file.txt    # each line becomes one element, no splitting
```

Bug: nested command substitution and quoting

```
# Outer $() creates a subshell; inner quotes work independently
result=$(echo "$(date '+%Y-%m-%d')")   # both levels of quoting are independen
t

# Backtick nesting requires escaping inner quotes — avoid backticks
old=`echo \`date\``   # ugly; $() is always clearer
```

Bug: arithmetic in an unquoted assignment splits the result

```
x=$(( 1 + 2 ))    # safe — arithmetic result is an integer, no spaces
# But consider:
IFS=2
x=$(( 1 + 2 ))    # result '3' — fine, IFS=2 only matters for unquoted expansi
ons
echo $x           # could split on '2' if result were '12' — quote it
```

Exercises

Exercise 1 — Expansion order detective

For each of the following lines, predict exactly what will be printed (or whether it will error), then verify in your shell. Explain which expansion stage causes the behaviour.

```
a='*.sh'
echo $a
echo "$a"
echo ${a/sh/py}
b='PATH'
echo ${!b}
set -- $'one\ntwo\nthree'
echo $#
```

```
# echo $a       — parameter expansion → '*.sh'; then word splitting (no spaces)
#               → then globbing: if .sh files exist, lists them; else prints '*.sh'
# Stage 7 (globbing) is the key stage.

# echo "$a"     — quoted: parameter expansion → '*.sh', no word splitting, no globbing
#               → prints literally:  *.sh

# echo ${a/sh/py}  — pattern substitution during parameter expansion (stage 3)
#                  → '*.py'; then word splitting (nothing to split)
#                  → then globbing: expands to .py files if any, else '*.py'

b='PATH'
echo "${!b}"
# Indirect expansion (stage 3): ${!b} looks up the variable whose name is in b
# → expands to the value of $PATH

set -- $'\none\ntwo\nthree'
echo $#
# $'...' is resolved by the LEXER (before any expansion stage)
# The result is a string with embedded newlines.
# set -- $'one\ntwo\nthree' — the $'...' becomes a single unquoted word
# with newlines inside it. Word splitting (stage 6) splits on newlines
```

```
# (newline is in IFS by default), giving three positional parameters.
# $# == 3
```

Exercise 2 — IFS surgery

Write a function `split_by SEP STRING RETARRAY` that splits STRING on the single-character delimiter SEP and stores the result in the array named RETARRAY — without using any external commands (no `awk`, `cut`, `tr`). Use IFS manipulation and the knowledge from this chapter about word splitting and namerefs.

```
split_by() {
  # Args: SEP STRING RETARRAY
  local _sep="$1"
  local _str="$2"
  local -n _ret="$3"

  # Save and set IFS locally — local IFS is scoped to this function
  local IFS="$_sep"

  # Word splitting fires when we assign the unquoted expansion to the arr
ay.
  # The read -ra trick is cleaner and handles empty fields:
  read -ra _ret <<<"$_str"
}

# Test
declare -a parts
split_by ':' 'one:two:three' parts
printf '[%s]\n' "${parts[@]}"
# [one]
# [two]
# [three]

split_by ',' 'a,,b,c' parts   # empty fields
printf '[%s]\n' "${parts[@]}"
# Note: read -ra collapses consecutive delimiters (IFS whitespace rules)
```

```
# For truly empty-field-preserving splits, use a while/case loop with ${s
tr%%SEP*}
```

Exercise 3 — Glob-safe pattern passing

Write a function `safe_find DIR PATTERN` that uses `find` to locate files matching PATTERN in DIR. The function must work correctly regardless of whether the current directory contains files that match PATTERN — i.e., it must prevent premature glob expansion. Also handle the case where DIR does not exist, exiting with a clear message.

```
safe_find() {
  local dir="${1:?safe_find: DIR required}"
  local pattern="${2:?safe_find: PATTERN required}"

  if [[ ! -d "$dir" ]]; then
    printf 'safe_find: not a directory: %s\n' "$dir" >&2
    return 1
  fi

  # Always quote $pattern so the shell never expands it.
  # find receives the literal string and does its own pattern matching.
  find "$dir" -type f -name "$pattern"
}

# Usage — safe even when *.log files exist in the current directory
safe_find /var/log '*.log'
safe_find /nonexistent '*.txt'   # prints error, returns 1
```

Exercise 4 — Expansion order puzzle: build a general renderer

Write a function `render_template TEMPLATE_FILE` that reads a file where `{{VAR}}` placeholders should be replaced with the values of shell variables of the same name. Requirements:

- Must NOT use `eval` (injection risk)
- Must handle multi-line templates
- Must leave `{{UNDEFINED}}` placeholders intact if the variable is not set

- Should use only Bash built-ins — no `sed` or `awk`

Hint: think about how parameter expansion's `${!varname}` indirect form and `${var:-default}` combine here.

```
render_template() {
  local file="${1:?template file required}"
  [[ -f "$file" ]] || { printf 'render_template: not found: %s\n' "$file"
>&2; return 1; }

  local line out rest varname value

  while IFS= read -r line || [[ -n $line ]]; do
    out=''
    rest="$line"

    # Scan for {{...}} placeholders using parameter expansion — no extern
al tools
    while [[ $rest == *'{{'*'}}'* ]]; do
      # Everything before the first {{
      out+="${rest%%\{\{*}"
      # Everything from {{ onward
      rest="${rest#*\{\{}"
      # The variable name is before }}
      varname="${rest%%\}\}*}"
      # Remainder after }}
      rest="${rest#*\}\}}"

      # Validate: varname must be a legal identifier
      if [[ $varname =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]]; then
        # Indirect expansion — ${!varname} retrieves the value
        value="${!varname}"
        if [[ -v $varname ]]; then
          # Variable is set — substitute its value
          out+="$value"
        else
          # Variable not set — leave placeholder intact
          out+="{{$varname}}"
        fi
      else
```

```
        # Not a valid identifier — leave it as-is
        out+="{{$varname}}"
      fi
    done

    printf '%s\n' "${out}${rest}"
  done < "$file"
}

# Test
NAME='Alice'
HOST='prod-01'
# Template file contains:
#   Hello {{NAME}}, deploying to {{HOST}}.
#   Token: {{SECRET}}   <-- SECRET not set
render_template template.txt
Hello Alice, deploying to prod-01.
Token: {{SECRET}}
```

Chapter 2 — POSIX Compliance and Portability

Every feature in the preceding chapters relies on Bash being the interpreter. The moment your script lands on Alpine Linux (which ships BusyBox ash), a minimal Docker image, a FreeBSD jail, or a macOS CI runner, Bash extensions silently fail or behave differently. This chapter maps the boundary between portable shell and Bash-specific syntax, shows you how to audit your scripts, and teaches you to write sh-compatible code when portability genuinely matters.

1 — The Shebang Line Is a Contract

The shebang declares the interpreter and sets expectations for every line that follows.

```
#!/bin/bash       # explicit bash — use all features freely, but requires bash at /bin/bash
#!/usr/bin/env bash  # portable bash — finds bash on PATH; preferred for distributed scripts
#!/bin/sh         # POSIX sh — must avoid ALL bash extensions
#!/usr/bin/env sh   # portable POSIX sh — on some systems sh IS bash (in POSIX mode)
```

> **`/bin/sh` is not Bash.** On Debian/Ubuntu, `/bin/sh` is **dash** — a fast, minimal POSIX-only shell. On macOS (since Catalina), `/bin/sh` is a locked-down build of Bash 3.2 in POSIX mode. On Alpine Linux and BusyBox systems it is **ash**. Writing `#!/bin/bash` and accidentally running it with `sh myscript.sh` ignores the shebang entirely — the shell is whatever `sh` resolves to.

Bash's POSIX mode

```
# Invoke bash in strict POSIX mode to test portability
bash --posix myscript.sh

# Or set from within a script (rarely useful in production)
set -o posix
```

2 — The Master Compatibility Table

These are the features you reach for automatically in Bash that simply do not exist in POSIX `sh`.

Feature	Bash syntax	POSIX / portable alternative
Arrays	`arr=(a b c)`, `${arr[@]}`	Positional params `set --`, or delimited strings
Associative arrays	`declare -A map`	No equivalent — use awk, or encode as `key=value` lines
Double-bracket test	`[[ expr ]]`	`[ expr ]` (quote everything) or `case`
Arithmetic compound cmd	`(( expr ))`	`[ $((expr)) -ne 0 ]` or `expr`
Process substitution	`<(cmd)`, `>(cmd)`	Named pipe (mkfifo) or temp file
Brace expansion	`{a,b,c}`, `{1..5}`	Explicit list or loop
Here-string	`<<< "string"`	`printf '%s' "string" |` or `echo … |`
`local` keyword	`local var=val`	POSIX optional but widely supported; use carefully
String case conversion	`${var^^}`, `${var,,}`	`printf '%s' "$var" | tr '[:lower:]' '[:upper:]'`
Regex matching	`[[ $x =~ regex ]]`	`printf '%s' "$x" | grep -Eq 'regex'` or `expr`
`mapfile` / `readarray`	`mapfile -t arr < file`	`while IFS= read -r line; do …`
Coprocesses	`coproc NAME { … }`	Named pipes (mkfifo)
`declare` / `typeset`	`declare -i`, `declare -r`	`readonly` for constants; no integer type
Extended globs	`!(*.log)`, `+(foo)`	Multiple explicit patterns or `case`
`$'...'` quoting	`$'\n'`, `$'\t'`	`printf '\n'` or `$(printf '\t')`
Source with arguments	`source file arg1`	`. file` (no arguments in strict POSIX)
`pipefail`	`set -o pipefail`	Not available — capture each pipe stage manually

Namerefs	`local -n ref=var`	No equivalent — use eval (carefully) or global names

3 — ShellCheck: Automated Portability Auditing

`shellcheck` is the single most valuable tool for portability work. It understands both the shebang and a `--shell` override, and distinguishes between bugs, style issues, and portability problems.

```
# Audit a script for POSIX compatibility
shellcheck --shell=sh myscript.sh

# Check for issues at a specific POSIX standard level
shellcheck --shell=sh --severity=warning myscript.sh

# In CI — exit non-zero on any warning or error
shellcheck -x --shell=sh myscript.sh

# Check multiple files at once
find . -name '*.sh' -print0 | xargs -0 shellcheck

# Inline directive to suppress a specific warning
# shellcheck disable=SC2039
local x='value'   # SC2039: 'local' not defined in POSIX sh
```

Key SC codes for portability

Code	Issue
SC2039	Feature not supported in `--posix` mode (`local`, `[[`, arrays, etc.)
SC3000–SC3999	Not POSIX (the SC3xxx series was added specifically for sh portability)
SC2006	Backtick command substitution — use `$(…)`
SC2196	`egrep` deprecated — use `grep -E`
SC2154	Variable referenced but not assigned
SC2086	Double-quote to prevent word splitting / globbing

4 — Writing POSIX-Compatible Code

The discipline is: reach for the portable form first, and use Bash extensions only when there is a genuine reason. Here are the most common translation patterns.

Replacing `[[ ]]` with `[ ]` and `case`

```
## Bash ──────────────────────────────────────────
[[ $str == foo* ]]              # pattern match
[[ $str =~ ^[0-9]+$ ]]          # regex match
[[ -n $x && -f $file ]]         # compound test

## POSIX ─────────────────────────────────────────
case "$str" in foo*) echo match ;; esac   # pattern via case

case "$str" in                              # digit-only check
  *[!0-9]*|'') echo "not integer" ;;
  *) # is integer
esac

[ -n "$x" ] && [ -f "$file" ]               # separate [ ] calls
```

Replacing arrays with positional parameters

```
## Bash ──────────────────────────────────────────
items=( one two three )
for item in "${items[@]}"; do echo "$item"; done

## POSIX ─────────────────────────────────────────
set -- one two three
for item in "$@"; do echo "$item"; done

# When you need to preserve $@ for the original arguments,
# use a wrapper function — each function has its own $@
process_list() {
  set -- one two three   # local to this function's positional params
```

```
  for item in "$@"; do echo "$item"; done
}
```

Replacing here-strings

```
## Bash
grep 'pattern' <<<"$variable"

## POSIX
printf '%s\n' "$variable" | grep 'pattern'
# or, if the variable might lack a trailing newline:
printf '%s' "$variable" | grep 'pattern'
```

Replacing `$'...'` quoting

```
## Bash
NL=$'\n'
TAB=$'\t'

## POSIX
NL=$(printf '\n')           # but trailing newline stripped! use sentinel:
NL=$(printf '\nx'); NL="${NL%x}"
TAB=$(printf '\t')
```

Replacing `pipefail`

```
## Bash
set -o pipefail
result=$(generate | transform | finalise)

## POSIX — capture intermediate exit codes manually
tmp=$(mktemp)
generate > "$tmp"; rc1=$?
result=$(transform < "$tmp" | finalise); rc2=$?
rm -f "$tmp"
[ $rc1 -eq 0 ] && [ $rc2 -eq 0 ] || { echo "pipeline failed" >&2; exit 1; }
```

Portable function-local variables

```
# 'local' is not in POSIX but is universally supported in practice
# (dash, ash, ksh, zsh all support it). The shellcheck SC2039 warning
# is technically correct but pragmatically safe to suppress for most targets.
# If targeting strict POSIX: use unique global names with a function prefix.

_mylib_process() {
  # POSIX-strict: prefix all vars to avoid collisions instead of local
  _mylib_process_input="$1"
  _mylib_process_result=''
  # … work …
}
```

5 — Platform-Specific Traps

macOS / BSD differences

macOS ships BSD userland tools alongside its own `/bin/sh` (Bash 3.2 in POSIX mode). The GNU tools and BSD tools share many flags but differ in critical ways.

```
## sed -i behaves differently
# GNU sed (Linux):
sed -i 's/foo/bar/' file.txt         # in-place, no backup

# BSD sed (macOS):
sed -i '' 's/foo/bar/' file.txt       # requires empty-string extension argume
nt

# Portable workaround:
sed 's/foo/bar/' file.txt > file.txt.tmp && mv file.txt.tmp file.txt

## stat differs
# GNU stat (Linux):
stat --format='%s' file.txt

# BSD stat (macOS):
stat -f '%z' file.txt
```

```
# Portable — use wc -c or ls:
size=$(wc -c < file.txt)

## date -d is GNU-only
# GNU:
date -d '2 days ago' '+%Y-%m-%d'

# BSD/macOS:
date -v-2d '+%Y-%m-%d'

# Portable — use Python or perl if available:
python3 -c "from datetime import date,timedelta; print(date.today()-timedelta
(2))"

## grep -P (PCRE) is GNU-only
grep -P '\d+' file     # Linux only
grep -E '[0-9]+' file   # portable ERE — works everywhere

## echo -e / echo -n are not portable
echo -e "line1\nline2"   # implementation-defined: some echo ignore -e
printf 'line1\nline2\n'  # always correct
```

BusyBox / Alpine Linux

```
# BusyBox ash: limited parameter expansion, no arrays, no [[, no (())
# Also missing from common BusyBox builds:
#   - read -a (array read)
#   - printf %q
#   - jobs -l
#   - kill -l with signal names

# Test in BusyBox ash without leaving your machine:
docker run --rm -v "$PWD:/work" alpine ash /work/myscript.sh

# Or interactively:
docker run --rm -it -v "$PWD:/work" alpine sh
```

Detecting the platform at runtime

```
is_macos()  { [ "$(uname)" = 'Darwin' ]; }
is_linux()  { [ "$(uname)" = 'Linux'  ]; }
is_busybox() {
  busybox --help >/dev/null 2&1
}

# Portable sed in-place using a wrapper
sed_inplace() {
  local expr="$1" file="$2"
  if is_macos; then
    sed -i '' "$expr" "$file"
  else
    sed -i "$expr" "$file"
  fi
}

# Detect gnu vs bsd coreutils once, export for the rest of the script
if date --version >/dev/null 2&1; then
  GNU_DATE=1    # GNU date supports --version
else
  GNU_DATE=0    # BSD/macOS date does not
fi
```

6 — The Decision Framework: When to Use What

Portability has a cost — portable code is usually more verbose and sometimes slower. Make the decision deliberately:

Situation	Recommended shebang	Rationale
Personal tooling, development machine	`#!/usr/bin/env bash`	Use Bash freely; clarity over portability
Distributed tool, unknown target systems	`#!/usr/bin/env bash` + check Bash version	Require Bash explicitly; document the requirement
Docker base-image entrypoint, Alpine image	`#!/bin/sh`	ash is the only shell; no Bash available

System init script, /etc/profile.d/	`#!/bin/sh`	Must work before Bash may be installed
CI/CD pipeline shared across Linux + macOS	`#!/usr/bin/env bash`	Bash 5 on macOS via Homebrew; document the dep
Library meant to be sourced by others	No shebang; document minimum shell version	Caller controls the interpreter

Checking the Bash version at startup

```
#!/usr/bin/env bash
# Require bash 4.2+ for associative arrays and [[ -v ]]
if (( BASH_VERSINFO[0] < 4 ||
      ( BASH_VERSINFO[0] == 4 && BASH_VERSINFO[1] < 2 ) )); then
  printf '%s requires bash >= 4.2 (have %s)\n' \
    "$(basename "$0")" "$BASH_VERSION" >&2
  exit 1
fi

# macOS ships Bash 3.2 — this guard catches it cleanly
```

7 — A Portable Script Template

This template deliberately avoids every Bash extension. It works correctly under dash, ash, ksh88, and any POSIX-conformant shell.

```
#!/bin/sh
# ============================================================================
=
# portable_tool.sh — POSIX sh only, no Bash extensions
# Tested: dash 0.5, ash (BusyBox 1.36), ksh93
# ============================================================================
=
set -eu    # -e and -u are POSIX; -o pipefail is NOT

PROG="${0##*/}"    # basename without a fork
```

```
die() { printf '%s: %s\n' "$PROG" "$*" >&2; exit 1; }

usage() {
  printf 'Usage: %s [OPTIONS] FILE\n' "$PROG"
  printf '  -v  verbose\n  -h  help\n'
}

VERBOSE=0

while [ $# -gt 0 ]; do
  case "$1" in
    -v) VERBOSE=1 ;;
    -h) usage; exit 0 ;;
    --) shift; break ;;
    -*) die "unknown option: $1" ;;
    *)  break ;;
  esac
  shift
done

[ $# -eq 0 ] && { usage >&2; exit 1; }
[ -f "$1" ] || die "file not found: $1"

# Main logic — using only POSIX features
while IFS= read -r line; do
  [ "$VERBOSE" = "1" ] && printf 'DEBUG: %s\n' "$line" >&2
  printf '%s\n' "$line"
done < "$1"
```

8 — Portability Anti-Patterns to Avoid

```
## Anti-pattern 1: assuming /usr/bin/env exists
# On stripped-down embedded systems, env may not be at /usr/bin/env
#!/usr/bin/env bash   # fine for general use
#!/bin/bash          # more portable on Linux where bash is always at /bin/ba
sh

## Anti-pattern 2: parsing ls output
```

```
for f in $(ls *.txt); do # WRONG — breaks on spaces, special chars
for f in *.txt; do          # RIGHT — glob directly

## Anti-pattern 3: using echo for arbitrary data
echo "$variable"    # risky if variable starts with - or contains \n
printf '%s\n' "$variable"  # always correct

## Anti-pattern 4: [ $var == value ] (double equals in [)
[ "$var" == "value" ]   # == is not POSIX in [ ] — works in bash/dash but not
all
[ "$var" = "value" ]    # = is POSIX — always use this in [ ]

## Anti-pattern 5: unportable test flags
[ -a file1 -a file2 ]   # -a as compound operator is not reliable
[ -e file ]              # -e not in original POSIX (but in SUSv3+ and all mod
ern sh)
[ -f file ]              # -f is always safe

## Anti-pattern 6: backtick nesting
result=`echo \`date\``    # hard to read; escaping is fragile
result=$(echo $(date))   # $(()) nests cleanly — use this always

## Anti-pattern 7: word splitting on unquoted $@
bad()  { echo $@;  }    # loses quoting from caller
good() { echo "$@"; }    # preserves individual arguments
```

Exercises

Exercise 1 — ShellCheck audit and fix

The script below is written with Bash idioms but has the shebang `#!/bin/sh`. Run it through `shellcheck --shell=sh` and fix every portability issue. The fixed script must produce identical output under both `bash` and `dash`.

```
#!/bin/sh
items=(alpha beta gamma)
for item in "${items[@]}"; do
  if [[ $item == g* ]]; then
```

```
    echo "Found: $item"
  fi
done
result=$(echo "Total: ${#items[@]}")
```

```
#!/bin/sh
# Fixed: no arrays, no [[, using case for pattern match, using set -- for
list

set -- alpha beta gamma
_count=$#     # capture count before any further set --

for item in "$@"; do
  case "$item" in
    g*) printf 'Found: %s\n' "$item" ;;
  esac
done

result=$(printf 'Total: %d' "$_count")
printf '%s\n' "$result"

# Changes made:
# 1. arrays → set -- / "$@"
# 2. ${#items[@]} → captured $# before the loop
# 3. [[ $item == g* ]] → case statement
# 4. echo "..." → printf '%s\n' "..." (echo not portable with $result)
```

Exercise 2 — Portable platform detection library

Write a POSIX-compatible library `lib/platform.sh` (no Bash extensions) that detects and exports the following variables when sourced:

- `PLATFORM` — one of: `linux`, `macos`, `freebsd`, `unknown`
- `SED_INPLACE` — the correct arguments to achieve in-place sed editing on this platform (e.g. `-i` or `-i ''`)
- `STAT_SIZE` — a function `stat_size FILE` that prints a file's byte count portably
- `HAS_GNU_DATE` — `1` if GNU date is available, `0` otherwise

```sh
#!/bin/sh
# lib/platform.sh — POSIX sh only
[ -n "${_LIB_PLATFORM_LOADED:-}" ] && return 0
_LIB_PLATFORM_LOADED=1

PLATFORM='unknown'
case $(uname -s) in
  Linux)   PLATFORM='linux'   ;;
  Darwin)  PLATFORM='macos'   ;;
  FreeBSD) PLATFORM='freebsd' ;;
esac

# sed -i flag
case "$PLATFORM" in
  macos|freebsd) SED_INPLACE="-i ''" ;;
  *)             SED_INPLACE='-i'    ;;
esac

# Portable file size function
stat_size() {
  _f="${1:?stat_size: FILE required}"
  case "$PLATFORM" in
    linux)
      stat --format='%s' "$_f" ;;
    macos|freebsd)
      stat -f '%z' "$_f" ;;
    *)
      # Universal fallback: wc -c handles any POSIX system
      wc -c < "$_f" | tr -d ' ' ;;
  esac
}

# GNU date detection
HAS_GNU_DATE=0
date --version >/dev/null 2&1 && HAS_GNU_DATE=1

export PLATFORM SED_INPLACE HAS_GNU_DATE
```

Exercise 3 — Portable `pipefail` wrapper

Write a POSIX sh function `pipe_all CMD1 CMD2 CMD3` that:

- Executes `CMD1 | CMD2 | CMD3`
- Returns exit code 0 only if *all three* commands succeeded
- Prints which stage(s) failed to stderr
- Works in dash and ash — no `pipefail`, no arrays, no `[[`

Hint: named pipes (mkfifo) let you interpose between stages and capture each exit code.

```
pipe_all() {
  # Usage: pipe_all CMD1 CMD2 CMD3
  # Runs CMD1 | CMD2 | CMD3, fails if any stage fails.
  # Uses temp files to capture exit codes since POSIX sh has no pipefail.

  [ $# -eq 3 ] || { printf 'pipe_all: exactly 3 commands required\n' >&2;
return 1; }

  _cmd1="$1" _cmd2="$2" _cmd3="$3"

  # Temp files for inter-stage data and exit codes
  _t1=$(mktemp); _t2=$(mktemp)
  _rc1=$(mktemp); _rc2=$(mktemp); _rc3=$(mktemp)

  # Cleanup on any exit
  trap 'rm -f "$_t1" "$_t2" "$_rc1" "$_rc2" "$_rc3"' EXIT INT TERM

  # Stage 1
  $_cmd1 > "$_t1"; printf '%d' $? > "$_rc1"
  # Stage 2
  $_cmd2 < "$_t1" > "$_t2"; printf '%d' $? > "$_rc2"
  # Stage 3
  $_cmd3 < "$_t2"; printf '%d' $? > "$_rc3"

  _r1=$(cat "$_rc1") _r2=$(cat "$_rc2") _r3=$(cat "$_rc3")
  _ok=0

  [ "$_r1" = "0" ] || { printf 'pipe_all: stage 1 failed (exit %s)\n' "$_
r1" >&2; _ok=1; }
```

```
  [ "$_r2" = "0" ] || { printf 'pipe_all: stage 2 failed (exit %s)\n' "$_
r2" >&2; _ok=1; }
  [ "$_r3" = "0" ] || { printf 'pipe_all: stage 3 failed (exit %s)\n' "$_
r3" >&2; _ok=1; }

  return "$_ok"
}
```

Exercise 4 — Cross-platform install script

Write a POSIX sh install script `install.sh` for a fictional tool `mytool` that:

- Works on Linux (GNU coreutils), macOS (BSD tools), and Alpine (BusyBox)
- Detects whether the user is root; if not, installs to `~/.local/bin` instead of `/usr/local/bin`
- Copies the binary, sets it executable, and verifies the install location is on PATH
- Prints a clear warning (not an error) if the install dir is not on PATH
- Uses only POSIX-portable shell constructs — no Bash extensions

---

```
#!/bin/sh
set -eu

PROG="${0##*/}"
BINARY='./mytool'    # source binary in current directory

die()  { printf '%s: error: %s\n' "$PROG" "$*" >&2; exit 1; }
warn() { printf '%s: warning: %s\n' "$PROG" "$*" >&2; }
info() { printf '%s: %s\n' "$PROG" "$*"; }

# Source binary must exist
[ -f "$BINARY" ] || die "source binary not found: $BINARY"

# Choose install directory
if [ "$(id -u)" = '0' ]; then
  INSTALL_DIR=/usr/local/bin
else
  INSTALL_DIR="${HOME}/.local/bin"
fi
```

```
# Create install directory if absent
[ -d "$INSTALL_DIR" ] || mkdir -p "$INSTALL_DIR" ||
  die "cannot create directory: $INSTALL_DIR"

# Copy and make executable
cp "$BINARY" "$INSTALL_DIR/mytool" || die "copy failed"
chmod '755' "$INSTALL_DIR/mytool" || die "chmod failed"

info "Installed mytool to $INSTALL_DIR/mytool"

# Check if INSTALL_DIR is on PATH (POSIX: use case to search PATH compone
nts)
_found=0
_oldIFS="$IFS"
IFS=':'
for _dir in $PATH; do
  [ "$_dir" = "$INSTALL_DIR" ] && { _found=1; break; }
done
IFS="$_oldIFS"

if [ "$_found" = "0" ]; then
  warn "$INSTALL_DIR is not on your PATH"
  printf '       Add this to your shell profile:\n'
  printf '         export PATH="%s:$PATH"\n' "$INSTALL_DIR"
else
  info "mytool is ready: $(command -v mytool 2>/dev/null || printf '%s/my
tool' "$INSTALL_DIR")"
fi
```

Chapter 3 — Advanced Array Patterns

Bash arrays go far deeper than iteration and length. This chapter covers the patterns that turn arrays into real data structures: sparse indexing, simulated multidimensional layouts, associative-array techniques, set algebra, in-place sorting, and serialisation strategies that survive process boundaries. All examples assume Bash 4.2 or later.

1 — Indexed Array Internals: Sparse Arrays

Bash indexed arrays are *sparse* — indices need not be contiguous, and a large highest index does not allocate intermediate memory. This is different from arrays in most languages and enables some useful patterns but also some surprising traps.

```
# Sparse assignment — gaps are fine
arr[0]=first
arr[100]=hundredth
echo "${#arr[@]}"    # 2 — counts ELEMENTS, not highest index
echo "${!arr[@]}"   # 0 100 — the actual indices (keys)

# Length vs highest index trap
arr=( a b c )
arr[99]=z
echo "${#arr[@]}"    # 4  — four elements
echo "${arr[-1]}"    # z  — negative indices count from highest key (bash 4.3
+)

# Iterating sparse arrays safely — always use ${!arr[@]}
for i in "${!arr[@]}"; do
  printf '[%d] = %s\n' "$i" "${arr[$i]}"
done

# Dangerous: for (( i=0; i<${#arr[@]}; i++ )) skips sparse gaps
# Safe: iterate keys with ${!arr[@]}

# Compact a sparse array back to contiguous 0-based
compact=( "${arr[@]}" )   # re-index as 0,1,2,... — values preserved, gaps rem
oved
```

Unsetting and re-packing

```
arr=( a b c d e )
unset 'arr[2]'          # removes index 2 — array is now sparse: 0,1,3,4
echo "${arr[@]}"        # a b d e  — gap is invisible in values
echo "${!arr[@]}"       # 0 1 3 4

# Remove element by value — filter into a new array
array_remove_value() {
  local -n __arr="$1"
  local __target="$2"
  local -a __new=()
  local __elem
  for __elem in "${__arr[@]}"; do
    [[ $__elem != "$__target" ]] && __new+=( "$__elem" )
  done
  __arr=( "${__new[@]}" )
}

fruits=( apple banana cherry banana )
array_remove_value fruits banana
printf '%s\n' "${fruits[@]}"   # apple  cherry
```

2 — Simulating 2D Arrays

Bash has no multidimensional arrays. The standard workaround encodes two indices into a single key using a separator that cannot appear in the data.

Associative array with compound key

```
declare -A grid

# Store grid[row,col] using comma as separator
grid_set() {
  local -n __g="$1"
  __g["${2},${3}"]="$4"
}
```

```
grid_get() {
  local -n __g="$1"
  printf '%s' "${__g["${2},${3}"]}"
}

grid_set grid 0 0 'top-left'
grid_set grid 0 1 'top-right'
grid_set grid 1 0 'bottom-left'

grid_get grid 0 1   # top-right

# Iterate all cells: parse the compound key
for key in "${!grid[@]}"; do
  row="${key%%,*}"
  col="${key##*,}"
  printf 'grid[%s][%s] = %s\n' "$row" "$col" "${grid[$key]}"
done
```

Maps of arrays via name-prefix convention

When you need a true map-of-arrays (e.g. a graph's adjacency list), use a naming convention to synthesise array names dynamically.

```
# Adjacency list: graph_edges_NODE is an array of neighbours
graph_add_edge() {
  local from="$1" to="$2"
  local arrname="graph_edges_${from}"
  declare -ga "$arrname"    # declare global array if not yet existing
  local -n __edges="$arrname"
  __edges+=( "$to" )
}

graph_neighbours() {
  local arrname="graph_edges_${1}"
  local -n __edges="$arrname"
  printf '%s\n' "${__edges[@]}"
}

graph_add_edge A B
graph_add_edge A C
```

```
graph_add_edge B D
graph_neighbours A    # B  C
graph_neighbours B    # D
```

3 — Associative Array Techniques

Default values and the -v test

```
declare -A counts

# Increment a counter — works even if key doesn't exist yet
counter_inc() {
  local -n __map="$1"
  local __key="$2"
  (( __map["$__key"]++ )) || true   # || true: (()) exits 1 when result is 0
}

while IFS= read -r word; do
  counter_inc counts "$word"
done < words.txt

# Print sorted by count descending
for word in "${!counts[@]}"; do
  printf '%d\t%s\n' "${counts[$word]}" "$word"
done | sort -rn

# Test if a key exists (vs testing if value is empty)
[[ -v 'counts[hello]' ]] && echo "key exists"
# Note: quote the subscript to prevent glob expansion of the key
```

Copying and merging associative arrays

```
# Deep copy — no built-in; iterate and assign
assoc_copy() {
  local -n __src="$1" __dst="$2"
  local __k
```

```
  for __k in "${!__src[@]}"; do
    __dst["$__k"]="${__src[$__k]}"
  done
}

# Merge b into a — b wins on key conflicts
assoc_merge() {
  assoc_copy "$2" "$1"   # copy src (b) into dst (a) — overwrites existing key
s
}

declare -A defaults=( [color]=blue [size]=medium [weight]=light )
declare -A overrides=( [color]=red [extra]=yes )
declare -A config
assoc_copy  defaults  config
assoc_merge config    overrides
# config: color=red size=medium weight=light extra=yes
```

Inverting a map

```
assoc_invert() {
  local -n __src="$1" __dst="$2"
  local __k
  for __k in "${!__src[@]}"; do
    __dst["${__src[$__k]}"]="$__k"
  done
}

declare -A code_to_name=( [US]='United States' [DE]=Germany [JP]=Japan )
declare -A name_to_code
assoc_invert code_to_name name_to_code
echo "${name_to_code[Germany]}"   # DE
```

4 — Set Operations

Treating arrays as mathematical sets — union, intersection, and difference — is a common need when comparing lists of servers, features, or packages.

Union

```
array_union() {
  # Stores union of arrays $1 and $2 into array $3
  local -n __a="$1" __b="$2" __out="$3"
  declare -A __seen
  local __e
  __out=()
  for __e in "${__a[@]}" "${__b[@]}"; do
    if [[ ! -v '__seen[$__e]' ]]; then
      __out+=( "$__e" )
      __seen["$__e"]=1
    fi
  done
}
```

Intersection

```
array_intersect() {
  local -n __a="$1" __b="$2" __out="$3"
  declare -A __in_b
  local __e
  __out=()
  for __e in "${__b[@]}"; do __in_b["$__e"]=1; done
  for __e in "${__a[@]}"; do
    [[ -v '__in_b[$__e]' ]] && __out+=( "$__e" )
  done
}
```

Difference (A minus B)

```
array_diff() {
  local -n __a="$1" __b="$2" __out="$3"
  declare -A __in_b
  local __e
  __out=()
  for __e in "${__b[@]}"; do __in_b["$__e"]=1; done
```

```
  for __e in "${__a[@]}"; do
    [[ ! -v '__in_b[$__e]' ]] && __out+=( "$__e" )
  done
}

# Demonstration
installed=( curl wget git vim tmux )
required=( git vim jq fzf )

declare -a missing common all_tools
array_diff      required  installed  missing
array_intersect required  installed  common
array_union     required  installed  all_tools

printf 'Missing: %s\n'   "${missing[@]}"    # jq  fzf
printf 'Common:  %s\n'   "${common[@]}"     # git  vim
printf 'All:     %s\n'   "${all_tools[@]}"  # git  vim  jq  fzf  curl  wget  t
mux
```

> **Using a temporary `declare -A` inside a function:** the `-A` inside the set-operation functions is *local* because `declare` inside a function creates a local variable by default. The `-g` flag would make it global — do not add it here.

5 — In-Place Sorting

Bash provides no built-in sort for arrays. The idiomatic solution routes values through the external `sort` command; for pure-Bash sorting (no fork), use a simple insertion sort or use `readarray` + `sort`.

Sort via `sort` command — simple and fast

```
array_sort() {
  local -n __arr="$1"; shift
  # Pass remaining args to sort (e.g. -r -n -u)
  local -a __sorted
  mapfile -t __sorted < <(printf '%s\n' "${__arr[@]}" | sort "$@")
  __arr=( "${__sorted[@]}" )
```

```
}

names=( Charlie Alice Bob Delta Alice )
array_sort names           # ascending lexicographic
printf '%s\n' "${names[@]}"
# Alice Alice Bob Charlie Delta

array_sort names -ru       # reverse, unique
printf '%s\n' "${names[@]}"
# Delta Charlie Bob Alice

nums=( 10 3 200 45 7 )
array_sort nums -n         # numeric sort
printf '%s\n' "${nums[@]}"
# 3 7 10 45 200
```

Sort by a derived key (Schwartzian transform in Bash)

```
# Sort files by their size without calling stat repeatedly
sort_files_by_size() {
  local -n __arr="$1"
  local -a __tagged __sorted
  local __f
  # Tag each filename with its size: "SIZE\tFILENAME"
  for __f in "${__arr[@]}"; do
    __tagged+=( "$(stat -c '%s' "$__f" 2>/dev/null || echo 0)$'\t'$__f" )
  done
  # Sort numerically, then strip the tag
  mapfile -t __sorted < <(
    printf '%s\n' "${__tagged[@]}" | sort -n | cut -f2
  )
  __arr=( "${__sorted[@]}" )
}

files=( /etc/hostname /etc/passwd /etc/shells )
sort_files_by_size files
printf '%s\n' "${files[@]}"
```

Pure-Bash insertion sort (no fork)

```
array_isort() {
  # In-place insertion sort — O(n²), use only for small arrays (<100 elements)
  local -n __a="$1"
  local __i __j __key
  for (( __i=1; __i < "${#__a[@]}"; __i++ )); do
    __key="${__a[$__i]}"
    __j=$(( __i - 1 ))
    while (( __j >= 0 )) && [[ "${__a[$__j]}" > "$__key" ]]; do
      __a[$(( __j + 1 ))]="${__a[$__j]}"
      (( __j-- )) || true
    done
    __a[$(( __j + 1 ))]="$__key"
  done
}

words=( banana apple cherry date elderberry )
array_isort words
printf '%s\n' "${words[@]}"
# apple banana cherry date elderberry
```

6 — Array Serialisation and Deserialisation

Arrays exist only within a process. To pass them across process boundaries — into subshells, to other scripts, or into files — you must serialise them. There is no single universally correct format; choose based on what the data can contain.

Method 1: `declare -p` round-trip

```
# declare -p produces valid Bash syntax that can be re-sourced
arr=( "hello world" "foo's bar" 'line1\nline2' )
declare -p arr
declare -a arr=([0]="hello world" [1]="foo's bar" [2]=$'line1\\nline2')

# Save to file and restore in another script
declare -p arr > /tmp/arr.bash
# In another script:
source /tmp/arr.bash
```

```
# arr is now restored with identical contents

# For associative arrays — identical pattern
declare -A map=( [key1]=val1 [key2]="with spaces" )
declare -p map > /tmp/map.bash
```

> **Security:** never `source` serialised data from an untrusted origin. `declare -p` output is executable shell code. Use NUL-delimited or JSON serialisation instead when the data comes from outside.

Method 2: NUL-delimited (safe for arbitrary content)

```
# Serialise: print each element followed by NUL
array_save_nul() {
  local -n __a="$1"
  local __file="$2"
  printf '%s\0' "${__a[@]}" > "$__file"
}

# Deserialise: read NUL-delimited elements
array_load_nul() {
  local -n __a="$1"
  local __file="$2"
  __a=()
  while IFS= read -r -d '' __elem; do
    __a+=( "$__elem" )
  done < "$__file"
}

data=( "element with spaces" $'\twith\ttabs' "line1
line2" )
array_save_nul data /tmp/data.nul

declare -a restored
array_load_nul restored /tmp/data.nul
printf '[%s]\n' "${restored[@]}"   # identical to data
```

Method 3: passing arrays across subshells via environment

```
# Subshells inherit variables but NOT array contents directly.
# Use declare -p to export, then eval to import.

export_array() {
  local name="$1"
  local -n __ref="$1"
  # Export as a string variable containing the declare -p output
  export "_EXPORTED_${name}"="$(declare -p "$name")"
}

import_array() {
  local name="$1"
  local varname="_EXPORTED_${name}"
  eval "${!varname}"   # safe: we control the source (declare -p output)
}

myarr=( one two three )
export_array myarr
# In a child process (subshell, bash -c, etc.):
bash -c 'source /dev/stdin <<<"${_EXPORTED_myarr}"; printf "%s\n" "${myarr
[@]}"'
```

7 — Stack and Queue Patterns

```
# Stack (LIFO) — push to end, pop from end
stack=()

stack_push() { local -n _s="$1"; _s+=( "$2" ); }
stack_pop()  {
  local -n _s="$1"
  (( "${#_s[@]}" > 0 )) || { echo "stack empty" >&2; return 1; }
  # Return value via a nameref
  local -n _retval="$2"
  _retval="${_s[-1]}"
  unset '_s[-1]'
}
stack_peek() { local -n _s="$1" _r="$2"; _r="${_s[-1]}"; }
```

```
stack_push stack alpha
stack_push stack beta
stack_push stack gamma
declare top
stack_pop  stack top; echo "popped: $top"   # gamma
stack_pop  stack top; echo "popped: $top"   # beta

# Queue (FIFO) — push to end, dequeue from front
queue=()
queue_enqueue() { local -n _q="$1"; _q+=( "$2" ); }
queue_dequeue() {
  local -n _q="$1" _r="$2"
  (( "${#_q[@]}" > 0 )) || { echo "queue empty" >&2; return 1; }
  _r="${_q[0]}"
  _q=( "${_q[@]:1}" )   # slice off first element — O(n)
}

# Note: queue_dequeue is O(n) due to the array re-assignment.
# For high-throughput queues, track a head index instead:
_queue_head=0
queue_dequeue_fast() {
  local -n _q="$1" _r="$2"
  (( _queue_head < "${#_q[@]}" )) || { echo "queue empty" >&2; return 1; }
  _r="${_q[$_queue_head]}"
  unset '_q[$_queue_head]'
  (( _queue_head++ )) || true
}
```

Exercises

Exercise 1 — Frequency map and top-N

Write a function `top_n RETARRAY N FILE` that reads words from FILE (one per line), counts their frequency using an associative array, and stores the top N words (sorted by descending count) in RETARRAY as elements of the form `COUNT:WORD`. Use only Bash built-ins plus `sort` — no `awk`, `uniq`, or `cut`.

```
top_n() {
  local -n __ret="$1"
  local __n="$2"
  local __file="$3"
  declare -A __freq
  local __word

  [[ -f "$__file" ]] || { printf 'top_n: file not found: %s\n' "$__file"
>&2; return 1; }

  while IFS= read -r __word; do
    [[ -z "$__word" ]] && continue
    (( __freq["$__word"]++ )) || true
  done < "$__file"

  # Build "COUNT WORD" lines, sort numerically descending, take top N
  local -a __lines
  for __word in "${!__freq[@]}"; do
    __lines+=( "${__freq[$__word]} $__word" )
  done

  mapfile -t __ret < <(
    printf '%s\n' "${__lines[@]}" |
    sort -rn |
    head -n "$__n" |
    while read -r __cnt __w; do
      printf '%s:%s\n' "$__cnt" "$__w"
    done
  )
}

# Test
declare -a result
top_n result 5 /usr/share/dict/words   # or any word-per-line file
printf '%s\n' "${result[@]}"
```

Exercise 2 — Symmetric difference and subset test

Using the set-operation functions from section 4 as a starting point, implement:

- `array_symdiff A B RESULT` — elements in A or B but not both
- `array_is_subset A B` — returns 0 if every element of A is in B, else 1

Then demonstrate both with a server-role example: `desired=(nginx certbot logrotate)` and `installed=(nginx curl wget logrotate)`.

```bash
array_symdiff() {
  local -n __a="$1" __b="$2" __out="$3"
  declare -A __in_a __in_b
  local __e
  __out=()
  for __e in "${__a[@]}"; do __in_a["$__e"]=1; done
  for __e in "${__b[@]}"; do __in_b["$__e"]=1; done
  for __e in "${__a[@]}"; do
    [[ ! -v '__in_b[$__e]' ]] && __out+=( "$__e" )
  done
  for __e in "${__b[@]}"; do
    [[ ! -v '__in_a[$__e]' ]] && __out+=( "$__e" )
  done
}

array_is_subset() {
  local -n __a="$1" __b="$2"
  declare -A __in_b
  local __e
  for __e in "${__b[@]}"; do __in_b["$__e"]=1; done
  for __e in "${__a[@]}"; do
    [[ -v '__in_b[$__e]' ]] || return 1
  done
  return 0
}

# Demonstration
desired=( nginx certbot logrotate )
installed=( nginx curl wget logrotate )

declare -a diff
array_symdiff desired installed diff
```

```
printf 'Symmetric diff: %s\n' "${diff[@]}"
# certbot  curl  wget

if array_is_subset desired installed; then
  echo "All desired packages are installed"
else
  echo "Some desired packages are missing"
  declare -a missing
  array_diff desired installed missing   # from section 4
  printf 'Install: %s\n' "${missing[@]}"
fi
```

Exercise 3 — Serialisation round-trip

Write two functions `array_to_json ARRNAME` and `json_to_array JSON ARRNAME` that serialise an indexed array to a JSON array string and back. Requirements:

- Values may contain double quotes, backslashes, and newlines — escape them correctly
- No external tools except `printf` (for serialisation); use `jq` only for deserialisation
- Demonstrate a round-trip: original array → JSON string → restored array, verifying each element matches

---

```
array_to_json() {
  local -n __a="$1"
  local __out='[' __sep='' __e __escaped
  for __e in "${__a[@]}"; do
    # Escape: backslash first, then double-quote, then control chars
    __escaped="${__e//\\/\\\\}"         # \ → \\
    __escaped="${__escaped//\"/\\\"}"   # " → \"
    # Replace literal newlines and tabs with JSON escapes
    __escaped="${__escaped//$'\n'/\\n}"
    __escaped="${__escaped//$'\t'/\\t}"
    __out+="${__sep}\"${__escaped}\""
    __sep=','
  done
  printf '%s]\n' "$__out"
}
```

```bash
json_to_array() {
  local __json="$1"
  local -n __a="$2"
  # jq -r outputs one element per line; handle newlines in values via NUL
  mapfile -t -d '' __a < <(printf '%s' "$__json" | jq -r '.[] | . + ""')
  # Strip the trailing empty element mapfile adds
  [[ "${__a[-1]}" == '' ]] && unset '__a[-1]'
}

# Round-trip test
original=(
  "simple"
  'with "quotes"'
  $'with\nnewline'
  $'back\\slash'
)

json=$(array_to_json original)
echo "JSON: $json"

declare -a restored
json_to_array "$json" restored

ok=true
for (( i=0; i < "${#original[@]}"; i++ )); do
  if [[ "${original[$i]}" != "${restored[$i]}" ]]; then
    printf 'MISMATCH at [%d]\n  orig: %q\n  rest: %q\n' \
      "$i" "${original[$i]}" "${restored[$i]}"
    ok=false
  fi
done
$ok && echo "Round-trip OK"
```

Exercise 4 — Dependency resolver

Using the adjacency-list pattern from section 2, implement a topological sort function `topo_sort GRAPH_PREFIX NODES_ARRAY RESULT_ARRAY` that:

- Takes a name prefix for adjacency arrays (e.g. `deps_` so that `deps_nginx=(openssl pcre)` means nginx depends on openssl and pcre)

- Performs a depth-first topological sort (dependencies before dependents)
- Detects cycles and prints an error, returning exit code 1
- Stores the sorted install order in RESULT_ARRAY

```
topo_sort() {
  local __prefix="$1"
  local -n __nodes="$2" __result="$3"
  declare -A __visited   # 0=unvisited 1=in-progress 2=done
  __result=()

  __dfs() {
    local __node="$1"
    case "${__visited[$__node]:-0}" in
      2) return 0 ;;                          # already processed
      1) printf 'topo_sort: cycle at %s\n' "$__node" >&2; return 1 ;;
    esac
    __visited["$__node"]=1   # mark in-progress

    # Visit dependencies first
    local __depname="${__prefix}${__node}"
    if [[ -v "$__depname" ]]; then
      local -n __deps="$__depname"
      local __dep
      for __dep in "${__deps[@]}"; do
        __dfs "$__dep" || return 1
      done
    fi

    __visited["$__node"]=2
    __result+=( "$__node" )
  }

  local __n
  for __n in "${__nodes[@]}"; do
    __dfs "$__n" || return 1
  done
}

# Define dependency graph via name-prefix arrays
deps_nginx=( openssl pcre zlib )
```

```
deps_certbot=( openssl python3 )
deps_python3=( zlib )
deps_openssl=( zlib )
deps_zlib=()
deps_pcre=()

packages=( nginx certbot )
declare -a install_order
topo_sort deps_ packages install_order
printf 'Install order:\n'
printf '  %s\n' "${install_order[@]}"
# zlib openssl pcre nginx python3 certbot
```

Chapter 4 — Coprocesses and Bidirectional IPC

Every time your script calls `$(command)` it forks a subshell, execs the command, collects the output through a pipe, and waits for the child to exit. For occasional calls that cost is negligible. For thousands of calls inside a tight loop it becomes the dominant bottleneck. Coprocesses and explicit bidirectional pipes give you a long-running peer process you can converse with — sending input and reading output repeatedly without ever re-forking the child.

1 — How coproc Works

The `coproc` keyword starts a command asynchronously and wires two anonymous pipes between it and the shell: one for the shell to write to the coprocess's stdin, and one to read from its stdout.

```
# Syntax forms
coproc NAME { command; }   # named coprocess — can be referenced by NAME
coproc      command        # unnamed — stored in array COPROC
```

After the `coproc` line runs, Bash exposes the pipe file descriptors in a two-element array:

Variable	Meaning
`NAME[0]`	FD open for reading (shell reads coprocess stdout)
`NAME[1]`	FD open for writing (shell writes coprocess stdin)
`$NAME_PID`	PID of the coprocess

```
# Minimal example: a coprocess running bc
coproc BC { bc -l; }

# Send an expression to bc's stdin
printf '%s\n' '3.14159 * 2' >&${BC[1]}

# Read bc's answer from bc's stdout
read -r answer <&${BC[0]}
```

```
echo "Result: $answer"   # Result: 6.28318

# Keep using it — no extra fork
printf '%s\n' 'sqrt(2)' >&${BC[1]}
read -r answer <&${BC[0]}
echo "Result: $answer"   # Result: 1.41421356

# Clean up: close pipes, wait for the process to exit
exec ${BC[1]}>&-          # close write end → coprocess sees EOF on its stdin
wait $BC_PID
```

The unnamed coprocess: `coproc command` without a name stores the FDs in `COPROC[0]` and `COPROC[1]`, and the PID in `$COPROC_PID`. Only one unnamed coprocess can exist at a time; a second `coproc` without a name silently replaces the first. Named coprocesses do not have this restriction.

2 — The Buffering Problem

The single most common coprocess failure is **deadlock** caused by output buffering. Most programs buffer their stdout when it is connected to a pipe (as opposed to a terminal). Your script writes a query, calls `read`, and blocks forever because the child's answer sits unflushed in its internal buffer.

```
# BROKEN: python's print() buffers stdout when stdout is a pipe
coproc PY { python3 -c '
import sys
for line in sys.stdin:
    sys.stdout.write("ECHO: " + line)   # BUFFERED — shell will hang
'; }

# FIX 1: force line-buffering with -u flag (Python-specific)
coproc PY { python3 -u -c '
import sys
for line in sys.stdin:
    sys.stdout.write("ECHO: " + line)
    sys.stdout.flush()
'; }

# FIX 2: stdbuf (coreutils) — works for any C stdio program
```

```
coproc GREP { stdbuf -oL grep 'ERROR'; }   # -oL = line-buffered stdout

# FIX 3: script -q (opens a pty, forcing line buffering)
coproc TOOL { script -q -c 'some-program' /dev/null; }

# FIX 4: expect's unbuffer utility
coproc TOOL { unbuffer -p some-program; }
```

Technique	Works for	Requires
`python3 -u`	Python only	nothing extra
`stdbuf -oL`	Any C stdio program	GNU coreutils
`unbuffer -p`	Any program (PTY trick)	expect package
`script -q -c`	Any program (PTY trick)	util-linux
Redesign with `\0` delimiter + `read -d ''`	Programs you control	nothing

3 — Synchronisation Sentinels

When you cannot change the child's buffering, use a sentinel: a known output line that marks the end of a response. Your script writes a query *plus* a command that generates the sentinel, then reads until it sees the sentinel.

```
# Pattern: send a query + "echo SENTINEL", read until SENTINEL appears
coproc SH { bash; }   # a subordinate shell as coprocess

sh_run() {
  local cmd="$1"
  local -a output
  local line
  local sentinel="__DONE_$$__"   # unique per PID

  # Send the command, then emit the sentinel via the subordinate shell
  printf '%s\necho %s\n' "$cmd" "$sentinel" >&${SH[1]}

  # Collect lines until we see the sentinel
  while IFS= read -r line <&${SH[0]}; do
```

```
    [[ $line == "$sentinel" ]] && break
    output+=( "$line" )
  done
  printf '%s\n' "${output[@]}"
}

sh_run 'ls /etc | head -3'
sh_run 'echo $HOSTNAME'

# Clean up
exec ${SH[1]}>&-
wait $SH_PID
```

4 — Explicit Bidirectional Pipes with exec

Before `coproc` was added in Bash 4.0, scripts used named FDs wired to process substitutions. This technique still works in all Bash versions and is often clearer when you need finer control over FD lifetimes.

```
# Open FDs 3 (read from child) and 4 (write to child) manually
exec 3<<(stdbuf -oL some-command)   # stdin of some-command not wired

# For true bidirectional wiring without coproc, use a FIFO
FIFO=$(mktemp -u)
mkfifo "$FIFO"

# Start the child with its stdin reading from the FIFO
stdbuf -oL some-command < "$FIFO" &
CHILD_PID=$!

# Open FD 4 for writing to the FIFO (child's stdin)
exec 4> "$FIFO"
rm "$FIFO"   # safe to unlink — FD 4 holds the write end open

# Write to child's stdin via FD 4
printf 'query\n' >&4

# Clean up
```

```
exec 4>&-
wait $CHILD_PID
```

> **Process substitution FD lifetimes:** a `<(cmd)` process substitution creates a pipe FD that is open only for the current command's scope. To keep it open across multiple operations, `exec N<<(cmd)` assigns it to a persistent FD number. Remember to close it with `exec N<&-` when done.

5 — Practical Coprocess Patterns

Pattern 1: Reusable database connection

```
# Open one persistent psql session instead of forking per query
coproc PSQL {
  psql --no-psqlrc --no-align --tuples-only \
       --field-separator='|' \
       -d "$DB_NAME" -U "$DB_USER"
}

_PSQL_SENTINEL="__PSQL_END_$$__"

db_query() {
  local sql="$1"
  local -n __rows="$2"
  local line
  __rows=()

  # Send SQL + a SELECT that emits the sentinel
  printf '%s;\nSELECT '"'"'%s'"'"';\n' \
    "$sql" "$_PSQL_SENTINEL" >&${PSQL[1]}

  while IFS= read -r line <&${PSQL[0]}; do
    [[ $line == "$_PSQL_SENTINEL" ]] && break
    [[ -n $line ]] && __rows+=( "$line" )
  done
}

declare -a users
```

```
db_query 'SELECT id, name FROM users WHERE active' users
for row in "${users[@]}"; do
  IFS='|' read -r id name <<< "$row"
  printf 'User %s: %s\n' "$id" "$name"
done

# Disconnect
exec ${PSQL[1]}>&-
wait $PSQL_PID
```

Pattern 2: Continuous log filter

```
# Use a coprocess to stream-filter a log with grep, then react in the shell
coproc FILTER { stdbuf -oL grep --line-buffered -E 'ERROR|WARN'; }

# Feed the log into the coprocess's stdin in the background
tail -f /var/log/app.log >&${FILTER[1]} &
TAIL_PID=$!

# React to each filtered line
while IFS= read -r line <&${FILTER[0]}; do
  printf '[ALERT] %s\n' "$line"
  # Could call send_slack_alert, page on-call, etc.
done

# Shutdown
kill $TAIL_PID
exec ${FILTER[1]}>&-
wait $FILTER_PID
```

Pattern 3: Long-running encoder / hash worker

```
# sha256sum reads one filename per line and emits HASH  FILENAME per line
# Re-use the process instead of forking for each file
coproc SHA { sha256sum -- $(cat); }

# Better: pipe filenames to sha256sum directly
# For line-by-line use, xargs is usually cleaner.
```

```
# But for a genuine conversation pattern, use a Python worker:

coproc HASHER {
  python3 -u -c '
import sys, hashlib
for path in sys.stdin:
    path = path.rstrip("\n")
    try:
        h = hashlib.sha256(open(path,"rb").read()).hexdigest()
        print(h, path, flush=True)
    except Exception as e:
        print("ERROR", path, str(e), flush=True)
'
}

hash_file() {
  local path="$1"
  printf '%s\n' "$path" >&${HASHER[1]}
  local result
  read -r result <&${HASHER[0]}
  printf '%s\n' "$result"
}

while IFS= read -r -d '' f; do
  hash_file "$f"
done < <(find /data -name '*.bin' -print0)

exec ${HASHER[1]}>&-
wait $HASHER_PID
```

6 — Multiple Coprocesses and Multiplexed IPC

Named coprocesses allow multiple long-running peers simultaneously. Each has its own FD array and PID variable.

```
# Two coprocesses running in parallel
coproc GEOCODER { python3 -u geocode_worker.py; }
coproc VALIDATOR { node validate_worker.js; }
```

```
# Send to each independently
geocode()  { printf '%s\n' "$1" >&${GEOCODER[1]};  read -r REPLY <&${GEOCODER
[0]}; }
validate() { printf '%s\n' "$1" >&${VALIDATOR[1]}; read -r REPLY <&${VALIDATOR
[0]}; }

geocode  "1600 Amphitheatre Parkway"
validate "user@example.com"

# Teardown helper — accept name, FD array, PID
coproc_close() {
  local wfd="$1"  pid="$2"
  exec {wfd}>&-    # Bash 4.1+: exec {varname}>&- closes the FD stored in varn
ame
  wait "$pid"
}
coproc_close "${GEOCODER[1]}"  $GEOCODER_PID
coproc_close "${VALIDATOR[1]}" $VALIDATOR_PID
```

7 — Bidirectional IPC Without coproc: FIFO Pairs

FIFOs are the portable alternative and are especially useful when the child process needs to be started by another mechanism (e.g. a service manager), or when you want multiple independent readers and writers.

```
bidir_open() {
  # Creates two FIFOs and stores the FD numbers in the given variable names
  local name="$1"       # name prefix for FIFOs
  local -n __rfd="$2"   # caller's variable to receive read FD
  local -n __wfd="$3"   # caller's variable to receive write FD
  local -n __pid="$4"   # caller's variable to receive child PID
  local fin="/tmp/${name}_in.$$"
  local fout="/tmp/${name}_out.$$"
  mkfifo "$fin" "$fout"
  # Start child reading from fin, writing to fout
  shift 4
  "$@" < "$fin" > "$fout" &
  __pid=$!
  exec {__rfd}< "$fout"
```

```
  exec {__wfd}> "$fin"
  rm -f "$fin" "$fout"   # unlink immediately — FDs keep the pipes alive
}

rfd wfd cpid
bidir_open worker rfd wfd cpid  stdbuf -oL ./worker.sh
printf 'hello\n' >&$wfd
read -r response <&$rfd
echo "Worker replied: $response"
exec {wfd}>&-; exec {rfd}<&-
wait "$cpid"
```

8 — Error Handling and Timeouts

```
# read -t N: read with timeout (fractional seconds supported)
coproc_read_timeout() {
  local fd="$1"
  local timeout="$2"
  local -n __out="$3"
  if IFS= read -r -t "$timeout" __out <&$fd; then
    return 0
  else
    local rc=$?
    if (( rc > 128 )); then
      printf 'coproc_read_timeout: timed out after %ss\n' "$timeout" >&2
      return 1
    else
      # rc=1: EOF — coprocess has exited
      printf 'coproc_read_timeout: coprocess exited\n' >&2
      return 2
    fi
  fi
}

# Check if a coprocess is still alive before sending
coproc_alive() {
  local pid="$1"
  kill -0 "$pid" 2>/dev/null
```

```
}

# Wrapper with alive-check, write, and timed read
coproc_call() {
  local wfd="$1" rfd="$2" pid="$3" request="$4" timeout="${5:-5}"
  local -n __reply="$6"
  coproc_alive "$pid" || { echo "coproc_call: process dead" >&2; return 1; }
  printf '%s\n' "$request" >&$wfd || { echo "coproc_call: write failed" >&2; r
eturn 1; }
  coproc_read_timeout "$rfd" "$timeout" __reply
}
```

9 — Performance: When Coprocesses Pay Off

A fork+exec costs roughly 1–5 ms on a modern Linux system. The payoff threshold for a coprocess is therefore around 50–100 calls; below that, the code complexity is not worth it.

```
# Benchmark: 1000 SHA256 calls — subprocess fork vs coprocess
declare -a files
mapfile -t -n 1000 files < <(find /usr/lib -name '*.so' -print)

# Method A: subprocess per file
TIMEFORMAT='A (subproc): %Rs'
time {
  for f in "${files[@]}"; do
    sha256sum "$f"
  done >/dev/null
}

# Method B: all files via a single xargs invocation (not a coprocess, but fai
r)
TIMEFORMAT='B (xargs):  %Rs'
time { printf '%s\0' "${files[@]}" | xargs -0 sha256sum >/dev/null; }

# Method C: coprocess Python worker
# (as shown in pattern 3 above)
# Typical ratio on a fast machine:
#   A (subproc): ~5.2s   — 1000 forks
```

```
#   B (xargs):   ~0.8s   — 1 fork, batched args
#   C (coproc):  ~0.4s   — 1 fork, 1000 round-trips over pipes
```

> **The sweet spot:** coprocesses shine when you need many *sequential* request-response exchanges with a single long-running process. If you want *parallel* execution, use `&` + `wait` or GNU parallel instead — those are covered in chapter 5. A coprocess is not a thread pool.

Exercises

Exercise 1 — Persistent `bc` calculator

Write a function `calc EXPR` that evaluates a `bc -l` expression and prints the result. The coprocess must be started on first use and reused for all subsequent calls (lazy initialisation via a flag variable). Calling `calc_close` should cleanly shut it down. Handle the case where the coprocess has died unexpectedly — restart it transparently.

```
_CALC_STARTED=0

_calc_start() {
  coproc _BC { bc -l; }
  _CALC_STARTED=1
}

calc() {
  local expr="$1"
  local result

  # Start or restart coprocess if needed
  if (( _CALC_STARTED == 0 )) || ! kill -0 "$_BC_PID" 2>/dev/null; then
    _calc_start
  fi

  # bc prints the result followed by a newline — one read suffices
  printf '%s\n' "$expr" >&${_BC[1]}
  IFS= read -r -t 5 result <&${_BC[0]} || {
    printf 'calc: timed out or EOF\n' >&2; return 1
```

```
  }
  printf '%s\n' "$result"
}

calc_close() {
  (( _CALC_STARTED )) || return 0
  exec ${_BC[1]}>&-
  wait $_BC_PID 2>/dev/null
  _CALC_STARTED=0
}

# Usage
calc '2^10'               # 1024
calc 'scale=4; sqrt(2)'   # 1.4142
for i in {1..20}; do
  calc "$i * $i"
done
calc_close
```

Exercise 2 — Coprocess-based CSV validator

Write a function `csv_validate_stream INPUT_FILE` that processes a CSV file line by line. For each data row (skip the header), send the row to a Python coprocess that validates the fields and returns either `OK` or `ERR: reason`. Collect and print a summary of how many rows passed and failed. Use `stdbuf -oL` or `python3 -u` to prevent buffering deadlock.

```
csv_validate_stream() {
  local file="$1"
  [[ -f "$file" ]] || { echo "File not found: $file" >&2; return 1; }

  coproc _VAL {
    python3 -u -c '
import sys, re
EMAIL_RE = re.compile(r"^[^@]+@[^@]+\.[^@]+$")
for raw in sys.stdin:
    row = raw.rstrip("\n").split(",")
    if len(row) != 3:
        print(f"ERR: expected 3 fields, got {len(row)}", flush=True)
```

```
        continue
    name, email, age = row
    if not name.strip():
        print("ERR: name is empty", flush=True)
    elif not EMAIL_RE.match(email.strip()):
        print("ERR: invalid email", flush=True)
    elif not age.strip().isdigit():
        print("ERR: age not numeric", flush=True)
    else:
        print("OK", flush=True)
'

  }

  local ok=0 fail=0 line lineno=0 reply
  while IFS= read -r line; do
    (( lineno++ ))
    (( lineno == 1 )) && continue   # skip header
    printf '%s\n' "$line" >&${_VAL[1]}
    IFS= read -r -t 3 reply <&${_VAL[0]} || { echo "Validator timeout" >&
2; break; }
    if [[ $reply == "OK" ]]; then
      (( ok++ ))
    else
      (( fail++ ))
      printf 'Row %d: %s\n' "$lineno" "$reply"
    fi
  done < "$file"

  exec ${_VAL[1]}>&-
  wait $_VAL_PID
  printf '\nSummary: %d OK, %d FAILED\n' "$ok" "$fail"
}
```

Exercise 3 — FIFO-based worker pool (single worker)

Without using `coproc`, implement a bidirectional communication channel using two FIFOs (one for requests, one for responses). Start a worker script that reads one job per line and prints a result per line. Write a `job_submit JOB` function and a `job_result` function that reads the

response. Demonstrate submitting 5 jobs and collecting their results. Include a `worker_shutdown` function that sends a quit signal and waits.

```
#!/usr/bin/env bash
# worker.sh — process jobs from stdin, print results to stdout
while IFS= read -r job; do
  [[ $job == "QUIT" ]] && break
  # Example: reverse each line
  rev <<< "$job"
done
#!/usr/bin/env bash
# controller.sh
_REQ_FIFO="/tmp/worker_req.$$"
_RES_FIFO="/tmp/worker_res.$$"
_WORKER_PID=""

worker_start() {
  mkfifo "$_REQ_FIFO" "$_RES_FIFO"
  stdbuf -oL ./worker.sh < "$_REQ_FIFO" > "$_RES_FIFO" &
  _WORKER_PID=$!
  exec {_WREQ}> "$_REQ_FIFO"
  exec {_WRES}< "$_RES_FIFO"
  rm -f "$_REQ_FIFO" "$_RES_FIFO"
}

job_submit() {
  printf '%s\n' "$1" >&$_WREQ
}

job_result() {
  local -n _r="$1"
  IFS= read -r -t 5 _r <&$_WRES
}

worker_shutdown() {
  job_submit "QUIT"
  exec {_WREQ}>&-
  exec {_WRES}<&-
  wait "$_WORKER_PID"
}
```

```
worker_start

for word in hello world bash script example; do
  job_submit "$word"
  declare result
  job_result result
  printf '%s → %s\n' "$word" "$result"
done

worker_shutdown
```

Exercise 4 — Benchmark: subprocess vs coprocess

Write a script that measures the wall-clock time to perform 500 integer additions using two methods:

1. A fresh `$(( ))` expansion per iteration (no fork, but as a baseline)
2. A coprocess running `python3 -u -c '...'` that reads `A+B` expressions and prints results
3. A fresh `python3 -c 'print(...)'` subprocess per iteration

Use `TIMEFORMAT='%R'` and `{ time ...; } 2>&1` to capture elapsed time for each method. Print a comparison table.

```
#!/usr/bin/env bash
set -euo pipefail
N=500
TIMEFORMAT='%R'

# Method 1: pure Bash arithmetic — no fork
t1=$( { time {
  for (( i=0; i<N; i++ )); do
    r=$(( i + i * 2 ))
  done
}; } 2>&1 )

# Method 2: coprocess python3
coproc _PY {
```

```
  python3 -u -c '
import sys
for line in sys.stdin:
    a, b = map(int, line.split("+"))
    print(a + b, flush=True)
'
}
t2=$( { time {
  for (( i=0; i<N; i++ )); do
    printf '%d+%d\n' "$i" $(( i*2 )) >&${_PY[1]}
    IFS= read -r _ <&${_PY[0]}
  done
}; } 2>&1 )
exec ${_PY[1]}>&-
wait $_PY_PID

# Method 3: subprocess per call
t3=$( { time {
  for (( i=0; i<N; i++ )); do
    python3 -c "print($i + $i * 2)"
  done >/dev/null
}; } 2>&1 )

printf '\n%-30s %8s\n'   "Method ($N iterations)" "Seconds"
printf '%s\n' "$(printf '%.0s-' {1..40})"
printf '%-30s %8s\n' "1. Bash arithmetic"       "$t1"
printf '%-30s %8s\n' "2. Python3 coprocess"     "$t2"
printf '%-30s %8s\n' "3. Python3 subprocess"    "$t3"
```

Chapter 5 — Concurrency and Parallelism

Bash has no threads, but it has processes — and processes are cheap enough on Linux that genuine parallel execution is well within reach of a shell script. This chapter covers the full spectrum: basic background jobs, precise waiting semantics, bounded job pools, race-condition avoidance, and GNU parallel for when you need industrial- strength throughput.

1 — Background Jobs and `wait`

The basics

```
# & sends a command to the background; $! is its PID
sleep 2 &
PID1=$!
sleep 3 &
PID2=$!

# wait PID — block until that specific process exits, return its exit status
wait $PID1; echo "sleep 2 exited: $?"
wait $PID2; echo "sleep 3 exited: $?"

# wait (no args) — block until ALL background jobs finish
for host in server{1..8}; do
  ping -c1 -W1 "$host" &>/dev/null &
done
wait
echo "All pings done"
```

`wait -n` — first to finish (Bash 5.1+)

```
# wait -n returns as soon as any one background job exits
# The exit status is that of the completed job
declare -a pids
```

```
for url in "${urls[@]}"; do
  curl -sfo "/tmp/dl_$$_${#pids[@]}" "$url" &
  pids+=( $! )
done

# Harvest results as they complete
for (( i=0; i<"${#pids[@]}"; i++ )); do
  wait -n
  echo "A job finished with status $?"
done

# wait -n -p VAR (Bash 5.3+) — also stores the PID of the completed job
wait -n -p finished_pid
echo "PID $finished_pid just completed"
```

Capturing exit statuses from background jobs

```
# Pattern: store PIDs, then wait for each and record exit codes
declare -A job_pids   # job_name → PID
declare -A job_exit   # job_name → exit code

run_job() {
  local name="$1"; shift
  "$@" &
  job_pids["$name"]=$!
}

harvest_jobs() {
  local name pid
  for name in "${!job_pids[@]}"; do
    pid="${job_pids[$name]}"
    wait "$pid"
    job_exit["$name"]=$?
  done
}

run_job backup    rsync -a /data/  /backup/
run_job compress  gzip -k /tmp/report.csv
run_job notify    curl -s "$WEBHOOK" -d '{"text":"starting"}'
harvest_jobs
```

```
for name in "${!job_exit[@]}"; do
  printf '%s: exit %d\n' "$name" "${job_exit[$name]}"
done
```

2 — Bounded Parallelism: Job Pools

Firing all jobs at once saturates CPU and I/O. A bounded pool keeps exactly *N* jobs running simultaneously — new ones start as old ones finish.

Simple semaphore with `wait -n`

```
parallel_run() {
  # parallel_run MAX_JOBS CMD [ARGS...]
  # Reads newline-delimited work items from stdin and runs CMD ITEM for each,
  # keeping at most MAX_JOBS running simultaneously.
  local max="$1"; shift
  local running=0
  local item
  while IFS= read -r item; do
    "$@" "$item" &
    (( running++ ))
    if (( running >= max )); then
      wait -n    # block until any one job finishes
      (( running-- ))
    fi
  done
  wait   # drain remaining jobs
}

process_file() {
  # Example worker: compress a file
  gzip -k "$1" && printf 'compressed %s\n' "$1"
}

find /data -name '*.log' | parallel_run 4 process_file
```

FD-based semaphore (works in Bash 4.x without `wait -n`)

```
# A counting semaphore using a FIFO and FD slots.
# Each "token" is a byte in the FIFO; a worker reads one to acquire,
# writes one back when done.

sem_init() {
  # sem_init N SEMAPHORE_FD_VAR
  local n="$1"
  local -n __fd="$2"
  local fifo
  fifo=$(mktemp -u)
  mkfifo "$fifo"
  exec {__fd}<>"$fifo"   # open for read+write so the FIFO stays open
  rm "$fifo"
  # Pre-fill with N tokens (one byte each)
  local i
  for (( i=0; i<n; i++ )); do
    printf 'x' >&$__fd
  done
}

sem_acquire() { IFS= read -r -n1 _ <&$1; }   # blocks until a token is availab
le
sem_release() { printf 'x' >&$1; }              # returns a token

# Usage
SEM_FD
sem_init 4 SEM_FD

for item in "${items[@]}"; do
  sem_acquire "$SEM_FD"   # blocks if 4 jobs already running
  {
    process_file "$item"
    sem_release "$SEM_FD"
  } &
done
wait
exec {SEM_FD}>&-
```

3 — Collecting Output Safely

Background jobs write to the same stdout as the parent. Interleaved output is a common problem. The cleanest solutions: write to per-job temp files, or use a dedicated output file per job.

```
# Anti-pattern: interleaved output
for host in a b c d; do
  { echo "=== $host ==="; ssh "$host" 'uptime'; } &    # lines from different
hosts intermix
done
wait

# Better: each job writes to its own temp file
declare -A tmpfiles
for host in a b c d; do
  tmpfiles["$host"]=$(mktemp)
  { echo "=== $host ==="; ssh "$host" 'uptime'; } > "${tmpfiles[$host]}" &
done
wait

# Print results in submission order
for host in a b c d; do
  cat "${tmpfiles[$host]}"
  rm -f "${tmpfiles[$host]}"
done

# Best: mktemp in a trap-guarded tmpdir so cleanup is guaranteed
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

for host in a b c d; do
  { ssh "$host" 'uptime'; } > "${TMPDIR}/${host}" &
done
wait
for host in a b c d; do
  printf '=== %s ===\n' "$host"
  cat "${TMPDIR}/${host}"
done
```

4 — Race Conditions and Shared Resources

Background jobs share the parent's open file descriptors and environment but run in separate processes with separate address spaces. The classic race conditions are: concurrent writes to a shared file and read-modify-write on a counter.

Atomic appends with `flock`

```
# flock -x LOCKFILE CMD — exclusive lock around CMD
LOG=/tmp/parallel.log

safe_log() {
  # Append atomically — flock serialises concurrent writers
  flock -x "${LOG}.lock" \
    printf '[%s] %s\n' "$(date +%T)" "$*" >> "$LOG"
}

for i in {1..20}; do
  { safe_log "job $i started"; sleep 0.1; safe_log "job $i done"; } &
done
wait
```

Shared counters via a locked temp file

```
# Shared mutable state between processes requires a file + lock.
# In-memory variables are NOT shared — each child has its own copy.

COUNTER_FILE=$(mktemp)
echo 0 > "$COUNTER_FILE"

counter_inc() {
  (
    flock -x 9
    local n
    n=$(< "$COUNTER_FILE")
    echo $(( n + 1 )) > "$COUNTER_FILE"
  ) 9>"${COUNTER_FILE}.lock"
}
```

```
counter_get() { < "$COUNTER_FILE"; }

for i in {1..50}; do
  { counter_inc; } &
done
wait
echo "Final count: $(counter_get)"   # 50 — not a random number
rm -f "$COUNTER_FILE" "${COUNTER_FILE}.lock"
```

The subshell variable trap

```
# Variables set inside & subshells are invisible to the parent
result=""
{ result="hello"; } &
wait
echo "'$result'"   # '' — the assignment happened in a child process

# Solution: communicate via file, pipe, or temp file
tmpf=$(mktemp)
{ echo "hello" > "$tmpf"; } &
wait
result=$(< "$tmpf")
echo "'$result'"   # 'hello'
rm "$tmpf"
```

5 — Signal Propagation to Children

When your script receives SIGINT or SIGTERM, background jobs do not automatically die — the kernel sends SIGINT to the entire foreground process group, but `trap` in the parent does not automatically reach `&` children spawned before the trap was set. Be explicit.

```
declare -a CHILD_PIDS

cleanup() {
  printf '\nInterrupted — killing children\n' >&2
  # Kill all children in our tracking array
```

```
  local pid
  for pid in "${CHILD_PIDS[@]}"; do
    kill "$pid" 2>/dev/null
  done
  wait
  exit 130
}
trap cleanup INT TERM

for item in "${items[@]}"; do
  slow_process "$item" &
  CHILD_PIDS+=( $! )
done
wait

# Alternative: kill the entire process group
cleanup_group() {
  # kill -- -$$ sends signal to every process in this script's process group
  kill -- -$$ 2>/dev/null
  exit 130
}
```

6 — GNU Parallel

GNU `parallel` is the right tool when you need configurable parallelism, progress bars, retry logic, argument templating, or cross-host execution. It is not a Bash built-in but is available on all major distributions and is worth knowing.

Basic usage

```
# Run one job per CPU core (default)
parallel gzip ::: *.log

# Explicit job count
parallel -j8 gzip ::: *.log

# Read arguments from stdin
find /data -name '*.csv' | parallel -j4 process_csv {}
```

```
# {} is the input item; {.} strips extension; {/} basename; {//} dirname
find /data -name '*.flac' | \
  parallel ffmpeg -i {} -q:a 2 '{.}.mp3'

# Multiple input sources
parallel echo '{1} x {2}' ::: a b c ::: 1 2
# a x 1   a x 2   b x 1   b x 2   c x 1   c x 2
```

Advanced options

```
# --bar: progress bar  --eta: time estimate  --joblog: machine-readable log
parallel --bar --joblog /tmp/jobs.log -j4 process_file ::: "${files[@]}"

# --retries N: retry failed jobs up to N times
parallel --retries 3 -j4 curl -sfo '{/}' ::: "${urls[@]}"

# --keep-order: print results in submission order (not completion order)
parallel --keep-order -j8 sha256sum ::: "${files[@]}"

# --delay S: stagger job starts by S seconds (rate limiting)
parallel --delay 0.5 -j4 curl -s ::: "${api_calls[@]}"

# --halt now,fail=1: stop all jobs as soon as one fails
parallel --halt now,fail=1 -j8 validate_file ::: "${files[@]}"

# Run on multiple remote hosts via SSH
parallel --sshlogin server1,server2,server3 uptime

# Export a shell function for parallel to use
my_func() { echo "Processing: $1"; }
export -f my_func
parallel -j4 my_func ::: "${items[@]}"
```

Reading the joblog

```
# The joblog TSV format:
# Seq  Host  Starttime  JobRuntime  Send  Receive  Exitval  Signal  Command
```

```
awk -F'\t' 'NR>1 && $7 != 0 { print "FAILED:", $NF }' /tmp/jobs.log
# List only failed jobs for re-submission
awk -F'\t' 'NR>1 && $7 != 0 { print $NF }' /tmp/jobs.log |
  parallel --retries 5 -j2 {}
```

7 — Patterns: Fan-out / Fan-in

A common parallel pipeline: fan-out splits a work queue among N workers; fan-in collects results into a single ordered stream. Both stages can be implemented in pure Bash.

```
#!/usr/bin/env bash
# Fan-out/fan-in: process files in parallel, collect results in order
set -euo pipefail

JOBS=${JOBS:-$(( $(nproc) * 2 ))}
WORKDIR=$(mktemp -d)
trap 'rm -rf "$WORKDIR"' EXIT

# --- Fan-out: submit all jobs, write output to numbered temp files ---
seq_num=0
declare -a pids seqfiles

while IFS= read -r -d '' file; do
  outfile="${WORKDIR}/${seq_num}.out"
  seqfiles+=( "$outfile" )

  # Bounded: wait if at capacity
  if (( "${#pids[@]}" >= JOBS )); then
    wait -n
    # Remove any finished PIDs from the array
    declare -a live=()
    for p in "${pids[@]}"; do
      kill -0 "$p" 2>/dev/null && live+=( "$p" )
    done
    pids=( "${live[@]}" )
  fi

  # Dispatch worker
  { sha256sum "$file"; } > "$outfile" &
```

```
  pids+=( $! )
  (( seq_num++ ))
done < <(find /usr/lib -name '*.so' -print0)

wait   # drain remaining

# --- Fan-in: concatenate results in submission order ---
for outfile in "${seqfiles[@]}"; do
  cat "$outfile"
done
```

8 — xargs -P: Simple Parallelism Without Bash Loops

```
# xargs -P N runs up to N processes simultaneously
# Combined with -n1 (one arg per process), this is a simple job pool

find /data -name '*.gz' -print0 |
  xargs -0 -P8 -I{} gunzip -k {}

# With a shell function: export the function, then call bash -c
process_item() {
  convert "$1" -resize '800x600>' "${1%.png}_thumb.png"
}
export -f process_item

find /photos -name '*.png' -print0 |
  xargs -0 -P$(nproc) -I{} bash -c 'process_item "$@"' _ {}

# xargs -P exit-code caveat: xargs exits 0 even if some children fail (xargs <
4.8)
# With GNU xargs 4.8+: exits non-zero if any child exited non-zero
# Portable workaround: track failures in a temp file
FAIL_FILE=$(mktemp)
find /data -name '*.csv' -print0 |
  xargs -0 -P4 -I{} bash -c 'validate_csv "$1" || touch "$2"' _ {} "$FAIL_FIL
E"
[[ -s "$FAIL_FILE" ]] && { echo "Some files failed validation" >&2; exit 1; }
rm "$FAIL_FILE"
```

Technique	Bash version	Best for	Limitation
`&` + `wait`	Any	Fire-and-forget, known small set	No built-in rate limiting
`wait -n` pool	5.1+	Bounded concurrency, large input	Cannot retrieve which PID finished
FIFO semaphore	4.1+	Bounded concurrency on 4.x	Slightly more setup code
`xargs -P`	Any	Simple one-liner parallelism	No per-job exit code tracking
GNU parallel	Any (external)	Production pipelines, SSH, retries	External dependency

Exercises

Exercise 1 — Parallel URL checker

Write a script `check_urls.sh` that reads URLs from a file (one per line) and checks each with `curl -sIo /dev/null -w '%{http_code}'`. Run at most 8 checks concurrently. Print a summary line per URL: `200 OK https://...` or `404 FAIL https://...`. Output must appear in the original URL order regardless of completion order. Non-200 codes should also be written to `failed_urls.txt`.

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

URL_FILE="${1:?Usage: $0 URL_FILE}"
MAX_JOBS=8
WORKDIR=$(mktemp -d)
trap 'rm -rf "$WORKDIR"' EXIT

declare -a urls pids
mapfile -t urls < "$URL_FILE"

running=0
for i in "${!urls[@]}"; do
```

```
  url="${urls[$i]}"
  outf="${WORKDIR}/${i}"
  {
    code=$(curl -sIo /dev/null -w '%{http_code}' --max-time 10 "$url" 2>/
dev/null || echo "000")
    printf '%s %s\n' "$code" "$url" > "$outf"
  } &
  pids+=( $! )
  (( ++running ))
  if (( running >= MAX_JOBS )); then
    wait -n
    (( running-- ))
  fi
done
wait

# Fan-in: print in order, collect failures
for i in "${!urls[@]}"; do
  line=$(< "${WORKDIR}/${i}")
  code="${line%% *}"
  if [[ $code == "200" ]]; then
    printf '%s OK   %s\n' "$code" "${urls[$i]}"
  else
    printf '%s FAIL %s\n' "$code" "${urls[$i]}"
    printf '%s\n' "${urls[$i]}" >> failed_urls.txt
  fi
done
```

Exercise 2 — Locked shared counter

Write a script that launches 100 background subshells, each calling a `counter_inc` function that increments a shared counter stored in a file. Use `flock` to prevent lost updates. After all jobs complete, assert the final counter value is exactly 100 and print PASS or FAIL. Then repeat the test *without* the lock and show that the unprotected version produces a value less than 100 (a classic race).

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

```
run_test() {
  local use_lock="$1"
  local cfile
  cfile=$(mktemp)
  local lfile="${cfile}.lock"
  echo 0 > "$cfile"

  inc_locked() {
    (
      flock -x 9
      n=$(<"$1")
      printf '%d\n' $(( n + 1 )) > "$1"
    ) 9>"$2"
  }

  inc_unlocked() {
    n=$(<"$1")
    printf '%d\n' $(( n + 1 )) > "$1"
  }

  for _ in {1..100}; do
    if [[ $use_lock == "yes" ]]; then
      inc_locked "$cfile" "$lfile" &
    else
      inc_unlocked "$cfile" &
    fi
  done
  wait

  local final
  final=$(<"$cfile")
  rm -f "$cfile" "$lfile"

  if (( final == 100 )); then
    printf '[lock=%s] final=%d PASS\n' "$use_lock" "$final"
  else
    printf '[lock=%s] final=%d FAIL (race condition demonstrated)\n' \
      "$use_lock" "$final"
  fi
}
```

```
run_test yes   # should always print PASS
run_test no    # will usually print FAIL (race)
```

Exercise 3 — Bounded image converter

Write `convert_all.sh SRCDIR DSTDIR` that converts every `.png` in SRCDIR to a 800px-wide JPEG in DSTDIR using ImageMagick's `convert` command. Constraints:

- Run at most `$(nproc)` conversions concurrently
- Trap SIGINT/SIGTERM and kill all running children before exiting
- Print a progress line `[N/TOTAL] converting FILE` before each job starts
- At the end print how many succeeded and how many failed (non-zero exit)

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

SRCDIR="${1:?Usage: $0 SRCDIR DSTDIR}"
DSTDIR="${2:?Usage: $0 SRCDIR DSTDIR}"
MAX=$(nproc)
WORKDIR=$(mktemp -d)
mkdir -p "$DSTDIR"

declare -a CHILD_PIDS

cleanup() {
  local p
  for p in "${CHILD_PIDS[@]}"; do kill "$p" 2>/dev/null; done
  rm -rf "$WORKDIR"
  exit 130
}
trap 'rm -rf "$WORKDIR"' EXIT
trap cleanup INT TERM

mapfile -t -d '' files < <(find "$SRCDIR" -maxdepth 1 -name '*.png' -prin
t0)
total="${#files[@]}"
done_count=0
```

```
ok=0; fail=0
running=0

for i in "${!files[@]}"; do
  src="${files[$i]}"
  base=$(basename "$src" .png)
  dst="${DSTDIR}/${base}.jpg"
  statusf="${WORKDIR}/${i}"

  (( done_count++ ))
  printf '[%d/%d] converting %s\n' "$done_count" "$total" "$base"

  {
    if convert "$src" -resize '800x>' "$dst" 2>/dev/null; then
      echo ok > "$statusf"
    else
      echo fail > "$statusf"
    fi
  } &
  CHILD_PIDS+=( $! )
  (( ++running ))
  if (( running >= MAX )); then
    wait -n; (( running-- ))
  fi
done
wait

for i in "${!files[@]}"; do
  status=$(< "${WORKDIR}/${i}")
  [[ $status == ok ]] && (( ok++ )) || (( fail++ ))
done
printf '\nDone: %d succeeded, %d failed\n' "$ok" "$fail"
(( fail == 0 )) || exit 1
```

Exercise 4 — GNU parallel deep dive

Using GNU `parallel`, write a one-liner (or short pipeline) that:

1. Finds all `.log` files under `/var/log`
2. Counts the number of lines containing `ERROR` in each file (using `grep -c`)

3. Runs up to 6 jobs concurrently
4. Keeps output in the original file order
5. Writes a `joblog` to `/tmp/grep_jobs.log`
6. Retries failed jobs up to 2 times

Then write a second pipeline that reads the joblog and prints the filenames of any jobs that ultimately failed (non-zero exit after retries).

```
# Part 1: parallel grep with all requirements
find /var/log -name '*.log' -print0 |
  parallel \
    --null \
    --keep-order \
    -j6 \
    --joblog /tmp/grep_jobs.log \
    --retries 2 \
    'grep -c "ERROR" {} || true; printf "%s\n" {}'

# Explanation of each flag:
#   --null        : input items are NUL-delimited (matches find -print0)
#   --keep-order  : output in submission order, not completion order
#   -j6           : at most 6 concurrent jobs
#   --joblog      : write TSV job log to this file
#   --retries 2   : retry up to 2 times on non-zero exit
#   || true       : grep exits 1 when no matches — treat as 0 matches, no
t failure

# Part 2: report ultimately-failed jobs from the joblog
# Joblog columns (tab-separated):
#   Seq Host Starttime JobRuntime Send Receive Exitval Signal Command
awk -F'\t' '
  NR == 1 { next }        # skip header
  $7 != 0 {               # Exitval column
    cmd = $NF
    # Extract filename: everything after the last space in the command
    n = split(cmd, parts, " ")
    print parts[n]
  }
' /tmp/grep_jobs.log
```

```
# Simpler with grep+cut for quick inspection:
awk -F'\t' 'NR>1 && $7!=0' /tmp/grep_jobs.log |
  cut -f9 |
  sed 's/.*[[:space:]]//'
```

Chapter 6 — Performance and Profiling

Bash is not Python. It is not Go. But it is also not as slow as most people assume — *when written well*. The performance gaps that matter are almost always attributable to a small number of identifiable patterns: unnecessary forking, reading files line by line through subshells, and ignoring built-ins that do the same job faster. This chapter gives you the tools to measure first and the knowledge to fix the right things.

1 — Measuring: `time`, `TIMEFORMAT`, and `date +%s%N`

The `time` built-in

```
# time wraps any simple command or { compound command; }
time find /usr -name '*.so' >/dev/null
# real    0m0.412s
# user    0m0.089s
# sys     0m0.323s

# TIMEFORMAT controls the output format (only affects the bash built-in time,
# not /usr/bin/time)
TIMEFORMAT='%R seconds real'
time { sleep 0.3; }    # 0.300 seconds real

# Capture the output — time writes to stderr
TIMEFORMAT='%R'
elapsed=$( { time my_function; } 2>&1 >/dev/null )
# Note: >/dev/null discards stdout of my_function; 2>&1 captures stderr (time
output)

# TIMEFORMAT tokens:
#   %R  elapsed real time (seconds, 3 decimal places)
#   %U  user CPU time
#   %S  system CPU time
#   %P  CPU percentage  (%U + %S) / %R * 100
```

High-resolution timestamps in a loop

```
# date +%s%N gives nanoseconds since epoch (GNU date only)
t0=$(date +%s%N)
# ... work ...
t1=$(date +%s%N)
printf '%.3f ms\n' "$(( (t1 - t0) / 1000000 ))"

# Portable microsecond timer via EPOCHREALTIME (bash 5.0+)
# No fork — just a variable read
t0="$EPOCHREALTIME"
# ... work ...
t1="$EPOCHREALTIME"
# EPOCHREALTIME is a float like 1718000000.123456
# Use awk or bc for the subtraction (bash can't do float arithmetic)
awk "BEGIN { printf \"%.6f s\n\", $t1 - $t0 }"

# Micro-benchmark helper — runs CMD N times and reports average
bench() {
  local n="${1}"; shift
  local i
  TIMEFORMAT='%R'
  local total
  total=$( { time { for (( i=0; i<n; i++ )); do "$@" >/dev/null 2&1; done; };
} 2>&1 )
  awk "BEGIN { printf \"%d runs: %.3fs total, %.4fs avg\n\", $n, $total, $tota
l/$n }"
}

bench 100 date +%s       # 100 runs: 0.312s total, 0.0031s avg
bench 100 printf '%(%s)T' -1  # 100 runs: 0.001s total, 0.0000s avg
```

2 — The Cost of Forks

Every `$(…)` command substitution and every `|` pipeline forks a subshell. On a modern system each fork costs roughly 0.5–3 ms. That sounds tiny, but a loop running 1 000 iterations with one subshell each costs 0.5–3 seconds doing nothing but forking.

Operation	Typical cost	Notes

Fork + exec (`$(date)`)	1–3 ms	Creates child process, execs binary
Subshell only (`$(printf…)`)	0.3–1 ms	Fork without exec — printf is a built-in
`$((…))` arithmetic	<1 µs	No fork — evaluated in-process
`${var//pat/rep}` substitution	<1 µs	No fork — built-in pattern engine
`read -r` from file	~5 µs/line	Syscall per character by default; use `mapfile`
`mapfile -t arr < file`	~0.5 µs/line	Reads in larger chunks
Pipe (`cmd | cmd`)	0.2–1 ms setup	Two forks for the pipeline itself

```
# Concrete comparison: 1000 timestamps

# Slow: fork+exec per iteration
TIMEFORMAT='%R'
time {
  for (( i=0; i<1000; i++ )); do
    ts=$(date +%s)      # fork + exec /usr/bin/date
  done
}
# ~2.8s

# Fast: built-in printf format (bash 4.2+)
time {
  for (( i=0; i<1000; i++ )); do
    printf -v ts '%(%s)T' -1   # no fork at all
  done
}
# ~0.004s — 700× faster
```

3 — Replacing External Commands with Built-ins

A systematic audit of your hot path often reveals external commands that Bash can replace with zero forks.

External command	Bash built-in replacement	Notes

`echo "text"`	`printf '%s\n' "text"`	Both are built-ins; prefer printf for portability
`$(date +%s)`	`printf -v v '%(%s)T' -1`	Bash 4.2+; `$EPOCHSECONDS` in 5.0+
`$(date +%Y-%m-%d)`	`printf -v v '%(%Y-%m-%d)T' -1`	strftime format via printf
`$(basename "$f")`	`${f##*/}`	Parameter expansion
`$(dirname "$f")`	`${f%/*}`	Caveat: returns `$f` if no slash
`$(echo "$s" | tr a-z A-Z)`	`${s^^}`	Bash 4.0+ case conversion
`$(echo "$s" | tr A-Z a-z)`	`${s,,}`	Bash 4.0+
`$(expr "$n" + 1)`	`$(( n + 1 ))` or `(( n++ ))`	No fork for arithmetic
`n=$(wc -l < file)`	`mapfile -t a < file; n=${#a[@]}`	No fork; trades memory for speed
`$(cat file)`	`$(< file)`	Bash opens the file directly — no cat fork
`grep -c '' file`	see `mapfile` above	Or use awk in one pass
`$(pwd)`	`$PWD`	Always in sync with current directory
`$(whoami)`	`$USER` or `${USER:-$(id -un)}`	$USER set by PAM/login
`$(hostname)`	`$HOSTNAME`	Bash sets this at startup
`$(printf '%s' "$v" | wc -c)`	`${#v}`	Character count (not bytes for multibyte)

```
# Pattern expansion as sed/grep replacement

# Remove leading whitespace
s="   hello"
s="${s#"${s%%[! ]*}"}"   # trim leading spaces — no fork

# Check if string contains a substring — no grep
if [[ "$haystack" == *"$needle"* ]]; then echo "found"; fi
```

```
# Check if string starts with prefix
[[ "$s" == error:* ]]

# Replace first occurrence — no sed
s="${s/foo/bar}"

# Replace all occurrences
s="${s//foo/bar}"

# Strip ANSI escape codes (a common sed job) — pure bash regex
clean="$line"
while [[ "$clean" =~ $'\e'\[[0-9;]*m ]]; do
  clean="${clean//"${BASH_REMATCH[0]}"}"
done
```

4 — Reading Files Efficiently

```
# Slowest: subshell per line
while IFS= read -r line; do
  processed+=( $(transform "$line") )   # fork per line!
done < input.txt

# Faster: read entire file into array with mapfile, then loop in bash
mapfile -t lines < input.txt
for line in "${lines[@]}"; do
  # pure-bash transformation here
  processed+=( "${line^^}" )
done

# Fastest for text processing: delegate the whole job to a single awk/sed pass
# One fork, no per-line overhead, purpose-built for text
awk '{ print toupper($0) }' input.txt > output.txt

# Reading a small file into a variable — $(< file) vs $(cat file)
content=$(< config.txt)    # Bash opens fd directly — no cat fork
content=$(cat config.txt)  # fork + exec cat — avoidable

# mapfile with a callback (-C -c): process every N lines without building the
```

```
# entire array in memory first
process_chunk() {
  # $1 = index of first new element, $2 = value
  printf 'Processing index %d: %s\n' "$1" "$2"
}
mapfile -t -C process_chunk -c 1 arr < big_file.txt
```

5 — PS4 Profiling: Line-Level Tracing

Setting `PS4` to include timestamps and enabling `set -x` turns every executed line into a timed trace. This is crude but requires no external tools and works on any Bash version.

```
# Minimal: show line numbers
PS4='+${BASH_SOURCE}:${LINENO}: '
set -x
# ... script body ...
set +x

# Timed: prefix each line with microseconds elapsed since start
# Requires bash 5.0+ for EPOCHREALTIME
_PROF_START="$EPOCHREALTIME"
PS4='+ $(awk "BEGIN{printf \"%8.4f\",'"$EPOCHREALTIME"'-'"$_PROF_START"'}")s
${BASH_SOURCE##*/}:${LINENO}: '

# Caution: the awk in PS4 forks on every traced line — only enable for short
# sections or you'll skew the results significantly.

# Better approach: redirect xtrace to a dedicated FD, analyse after
exec 19>/tmp/trace_$$
BASH_XTRACEFD=19
PS4='+ $EPOCHREALTIME ${BASH_SOURCE##*/}:${LINENO}: '
set -x

# ... script body ...

set +x
exec 19>&-

# Analyse the trace: find lines that took the most time
```

```
# Each line of the trace looks like: + 1718034567.123456 script.sh:42: command
awk '
  /^\+/ {
    split($2, a, ".")
    ts = a[1] "." a[2]
    if (prev_ts != "") {
      elapsed = ts - prev_ts
      printf "%8.4fs  %s\n", elapsed, prev_line
    }
    prev_ts = ts
    prev_line = substr($0, index($0,$3))
  }
' /tmp/trace_$$ | sort -rn | head -20
```

> **BASH_XTRACEFD** (Bash 4.1+) redirects `set -x` output to a specific file descriptor instead of stderr. This lets you capture trace data without polluting the script's error stream, making it safe to enable in production for a single slow function.

6 — Function-Level Profiling with DEBUG Trap

```
# The DEBUG trap fires before every simple command.
# Combine with BASH_SOURCE / FUNCNAME / LINENO for call-graph data.

declare -A _PROF_ENTER   # function → entry timestamp
declare -A _PROF_TOTAL   # function → cumulative ms
declare -A _PROF_CALLS   # function → call count

_prof_debug() {
  local fn="${FUNCNAME[1]:-main}"
  local now="$EPOCHREALTIME"
  # Record entry on first call in this function frame
  [[ -z "${_PROF_ENTER[$fn]:-}" ]] && _PROF_ENTER["$fn"]="$now"
}

_prof_return() {
  local fn="${FUNCNAME[1]:-main}"
  local now="$EPOCHREALTIME"
  local entry="${_PROF_ENTER[$fn]:-}"
```

```
  [[ -z "$entry" ]] && return
  local ms
  ms=$(awk "BEGIN{printf \"%d\", ($now-$entry)*1000}")
  (( _PROF_TOTAL["$fn"]+=ms )) || true
  (( _PROF_CALLS["$fn"]++   )) || true
  unset '_PROF_ENTER[$fn]'
}

prof_start() {
  trap '_prof_debug' DEBUG
  trap '_prof_return' RETURN
}

prof_report() {
  printf '\n%-30s %8s %8s %10s\n' "Function" "Calls" "Total ms" "Avg ms"
  printf '%s\n' "$(printf '%.0s-' {1..60})"
  for fn in "${!_PROF_TOTAL[@]}"; do
    printf '%-30s %8d %8d %10.2f\n' \
      "$fn" \
      "${_PROF_CALLS[$fn]}" \
      "${_PROF_TOTAL[$fn]}" \
      "$(( _PROF_TOTAL["$fn"] * 100 / _PROF_CALLS["$fn"] ))e-2"
  done | sort -k3 -rn
}
```

7 — Hot-Path Anti-Patterns and Fixes

Anti-pattern 1: Forking inside a loop

```
# Bad: one fork per iteration
for f in "${files[@]}"; do
  ext=$(echo "$f" | sed 's/.*\.//')   # fork + exec echo + fork + exec sed
  base=$(basename "$f")                # fork + exec basename
done

# Good: parameter expansion — zero forks
for f in "${files[@]}"; do
  ext="${f##*.}"
```

```
  base="${f##*/}"
done
```

Anti-pattern 2: Useless use of cat

```
# Bad: cat is just a conduit
cat file.txt | grep "pattern"
cat file.txt | wc -l

# Good: redirect directly
grep "pattern" file.txt
wc -l < file.txt
```

Anti-pattern 3: Redundant subshells in conditionals

```
# Bad: grep returns an exit code — no need to capture output
if [[ $(grep -c "pattern" file) -gt 0 ]]; then echo found; fi

# Good: test the exit code directly
if grep -q "pattern" file; then echo found; fi

# Bad: wc in arithmetic
if (( $(wc -l < file) > 100 )); then echo big; fi

# Good: mapfile + array length if you need the data anyway,
# or awk for a single-pass threshold check (no array allocation)
awk 'END { exit (NR > 100 ? 0 : 1) }' file && echo big
```

Anti-pattern 4: Concatenating strings with `$(...)`

```
# Bad: three forks to build a path string
logfile="$(dirname "$script")/logs/$(basename "$script" .sh)_$(date +%Y%m%d).l
og"

# Good: parameter expansion + built-in printf strftime
dir="${script%/*}"
name="${script##*/}"; name="${name%.sh}"
```

```
printf -v stamp '%(%Y%m%d)T' -1
logfile="${dir}/logs/${name}_${stamp}.log"
```

Anti-pattern 5: Piping to while read

```
# Bad: the while loop runs in a subshell — variables set inside are lost
grep "ERROR" app.log | while IFS= read -r line; do
  (( errors++ ))   # invisible to parent
done
echo "errors: $errors"   # always 0

# Good: redirect with process substitution — loop runs in current shell
while IFS= read -r line; do
  (( errors++ )) || true
done < <(grep "ERROR" app.log)
echo "errors: $errors"   # correct

# Even better: let awk count in one pass — no while loop at all
errors=$(grep -c "ERROR" app.log)
```

8 — When to Stop Using Bash

Recognising the boundary where Bash becomes the wrong tool is as important as knowing how to optimise it. These signals suggest it is time to reach for Python, Go, or another language:

Signal	Reason to switch
Processing more than ~100k lines of text per run	awk/Python will be 10–100× faster with less code
Nested data structures (JSON, YAML, XML)	Bash has no native JSON parser; `jq` is a fork per call
More than ~500 lines of logic	No type system, no unit-testable modules, hard to maintain
Floating-point arithmetic throughout	Bash has no floats; every float operation needs awk or bc

HTTP client / API integration	Python requests or Go net/http are far more ergonomic
Stateful concurrency (shared queues, worker pools)	No threads; file-based synchronisation is brittle at scale
Security-sensitive string parsing	Too easy to introduce injection; use a real parser

> **The hybrid pattern:** the sweet spot is often a thin Bash wrapper that orchestrates the environment (PATH, config files, working directory) and hands off the heavy lifting to a Python or Go binary. The shell excels at *wiring*; compiled languages excel at *computing*.

Exercises

Exercise 1 — Fork audit

The following function has at least six unnecessary forks. Identify each one and rewrite the function to eliminate all of them using only Bash built-ins. Then benchmark both versions with `bench 200 original_func` and `bench 200 fast_func` and report the speedup ratio.

```
original_func() {
  local filepath="$1"
  local dir=$(dirname "$filepath")
  local base=$(basename "$filepath")
  local ext=$(echo "$base" | sed 's/.*\.//')
  local stem=$(basename "$base" ".$ext")
  local upper=$(echo "$stem" | tr 'a-z' 'A-Z')
  local ts=$(date '+%Y%m%d_%H%M%S')
  echo "${dir}/${upper}_${ts}.${ext}"
}
```

```
# Forks in original_func:
# 1. $(dirname)   → ${filepath%/*}
# 2. $(basename)  → ${filepath##*/}
# 3. $(echo | sed) to get ext → ${base##*.}
# 4. $(basename base .ext) to get stem → ${base%.*}
```

```bash
# 5. $(echo | tr) to uppercase → ${stem^^}
# 6. $(date) → printf -v ts '%(%Y%m%d_%H%M%S)T' -1

fast_func() {
  local filepath="$1"
  local dir="${filepath%/*}"
  local base="${filepath##*/}"
  local ext="${base##*.}"
  local stem="${base%.*}"
  local upper="${stem^^}"
  local ts; printf -v ts '%(%Y%m%d_%H%M%S)T' -1
  printf '%s/%s_%s.%s\n' "$dir" "$upper" "$ts" "$ext"
}

# Benchmark helper from section 1
bench() {
  local n="$1"; shift
  local i
  TIMEFORMAT='%R'
  local total
  total=$( { time { for (( i=0; i<n; i++ )); do
    "$@" >/dev/null 2&1
  done; }; } 2>&1 )
  awk "BEGIN { printf \"%-20s %d runs: %.3fs (%.4f avg)\n\", \"$1\", $n,
$total, $total/$n }"
}

bench 200 original_func "/var/log/app/server.log"
bench 200 fast_func     "/var/log/app/server.log"
# Typical output:
# original_func        200 runs: 2.461s (0.0123 avg)
# fast_func            200 runs: 0.003s (0.0000 avg)  ~800× faster
```

Exercise 2 — PS4 trace analyser

Write a script `profile_me.sh` that:

1. Opens FD 19 to `/tmp/trace_$$.log`
2. Sets `BASH_XTRACEFD=19` and a timestamped `PS4`

3. Enables `set -x` then runs a deliberately slow function (at least 5 different operations including a `date` call, a `grep`, and a `wc`)
4. Disables tracing and closes FD 19
5. Parses the trace log with awk to produce a table of the top 10 slowest lines (line content + elapsed time in ms since previous traced line)

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

TRACE_LOG="/tmp/trace_$$.log"
trap 'rm -f "$TRACE_LOG"' EXIT

# ---- Slow function under test ----
slow_work() {
  local ts
  ts=$(date '+%Y-%m-%d %H:%M:%S')                    # fork 1
  local count
  count=$(grep -c 'root' /etc/passwd)                 # fork 2
  local lines
  lines=$(wc -l < /etc/passwd)                        # fork 3
  local upper
  upper=$(echo "$ts" | tr 'a-z' 'A-Z')                # fork 4+5
  local host
  host=$(hostname)                                    # fork 6
  printf '%s | root lines: %d/%d | host: %s\n' \
    "$upper" "$count" "$lines" "$host"
}

# ---- Enable tracing to FD 19 ----
exec 19>"$TRACE_LOG"
BASH_XTRACEFD=19
PS4='+ $EPOCHREALTIME ${BASH_SOURCE##*/}:${LINENO}: '
set -x

slow_work

set +x
exec 19>&-
unset BASH_XTRACEFD
```

```
# ---- Analyse trace log ----
printf '\nTop 10 slowest lines:\n'
awk '
/^\+/ {
  # Fields: + TIMESTAMP SOURCE:LINE: rest...
  ts   = $2
  rest = ""
  for (i = 3; i <= NF; i++) rest = rest (i==3 ? "" : " ") $i

  if (prev_ts != "") {
    elapsed_ms = (ts - prev_ts) * 1000
    printf "%8.2f ms  %s\n", elapsed_ms, prev_rest
  }
  prev_ts   = ts
  prev_rest = rest
}
' "$TRACE_LOG" | sort -rn | head -10
```

Exercise 3 — File-reading speed comparison

Write a script that reads a moderately large file (generate one with `seq 1 50000 > /tmp/test.txt`) using four methods and times each:

1. `while IFS= read -r line; do ... done < file` — count lines with a bash counter
2. `mapfile -t arr < file` — then `${#arr[@]}`
3. `wc -l < file`
4. `awk 'END{print NR}' file`

Print a formatted comparison table showing elapsed time and lines-per-second for each method. All four must produce the same count.

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

FILE=/tmp/bench_read_$$.txt
seq 1 50000 > "$FILE"
trap 'rm -f "$FILE"' EXIT
```

```
TIMEFORMAT='%R'
declare -a labels times counts

# Method 1: while read
c1=0
t1=$( { time {
  while IFS= read -r _; do (( c1++ )) || true; done < "$FILE"
}; } 2>&1 )

# Method 2: mapfile
declare -a arr
t2=$( { time { mapfile -t arr < "$FILE"; }; } 2>&1 )
c2="${#arr[@]}"
unset arr

# Method 3: wc -l
t3=$( { time { c3=$(wc -l < "$FILE"); }; } 2>&1 )

# Method 4: awk
t4=$( { time { c4=$(awk 'END{print NR}' "$FILE"); }; } 2>&1 )

# Verify all agree
for c in "$c1" "$c2" "$c3" "$c4"; do
  [[ "$c" == "$c1" ]] || { echo "Count mismatch!" >&2; exit 1; }
done

# Print table
printf '\n%-25s %10s %8s %15s\n' "Method" "Lines" "Time(s)" "Lines/sec"
printf '%s\n' "$(printf '%.0s-' {1..62})"
while IFS='|' read -r label t c; do
  awk "BEGIN {
    t = $t; c = $c
    lps = (t > 0) ? int(c/t) : 999999
    printf \"%-25s %10d %8.4f %15d\n\", \"$label\", c, t, lps
  }"
done <<< $"while read|$t1|$c1
mapfile|$t2|$c2
wc -l|$t3|$c3
awk NR|$t4|$c4"
```

Exercise 4 — Hot-path rewrite

The script below processes a log file and is painfully slow on large inputs. Identify every performance problem, then rewrite it as a single `awk` program that produces identical output. Verify correctness on a 10,000-line synthetic log, then benchmark both versions.

```
#!/usr/bin/env bash
# slow_stats.sh — count ERROR/WARN/INFO lines and show top 3 error messag
es
errors=0; warns=0; infos=0
declare -A msg_count
while IFS= read -r line; do
  level=$(echo "$line" | cut -d' ' -f2)
  msg=$(echo "$line" | cut -d' ' -f3-)
  case "$level" in
    ERROR) (( errors++ )); (( msg_count["$msg"]++ )) || true ;;
    WARN)  (( warns++  )) ;;
    INFO)  (( infos++  )) ;;
  esac
done < "$1"
printf 'ERROR: %d  WARN: %d  INFO: %d\n' "$errors" "$warns" "$infos"
printf 'Top 3 error messages:\n'
for msg in "${!msg_count[@]}"; do
  printf '%d %s\n' "${msg_count[$msg]}" "$msg"
done | sort -rn | head -3
```

```
# Problems in slow_stats.sh:
# 1. while IFS= read loop — slow for large files (syscall per character)
# 2. Two $(echo | cut) forks per line — 2N forks total
# 3. The while|pipe anti-pattern: if it were cmd | while, vars would be l
ost
#    (here it's a redirect, so that's not the issue — but the forks are)
# Solution: one awk pass does everything — zero extra forks

# fast_stats.awk (inline here as a bash here-doc)
fast_stats() {
  awk '
```

```
  {
    level = $2
    msg   = substr($0, index($0, $3))
    if      (level == "ERROR") { errors++; msg_count[msg]++ }
    else if (level == "WARN")  { warns++  }
    else if (level == "INFO")  { infos++  }
  }
  END {
    printf "ERROR: %d  WARN: %d  INFO: %d\n", errors, warns, infos
    print "Top 3 error messages:"
    # Sort msg_count by value descending — build array of "count msg" lin
es
    n = 0
    for (m in msg_count) lines[n++] = sprintf("%06d %s", msg_count[m], m)
    # Simple insertion sort on lines[]
    for (i = 1; i < n; i++) {
      key = lines[i]; j = i - 1
      while (j >= 0 && lines[j] < key) { lines[j+1] = lines[j]; j-- }
      lines[j+1] = key
    }
    for (i = n-1; i >= n-3 && i >= 0; i--) {
      cnt = lines[i]+0; sub(/^[0-9]+ /, "", lines[i])
      printf "%d %s\n", cnt, lines[i]
    }
  }
  ' "$1"
}

# Generate a synthetic 10k-line log for testing
gen_log() {
  local levels=( ERROR WARN INFO INFO INFO )
  local msgs=( "disk full" "timeout" "connection refused" "auth failed"
"ok" )
  for (( i=0; i<10000; i++ )); do
    printf '2024-01-01 %s %s\n' \
      "${levels[RANDOM%5]}" "${msgs[RANDOM%5]}"
  done
}

LOGFILE=/tmp/test_log_$$.txt
trap 'rm -f "$LOGFILE"' EXIT
```

```
gen_log > "$LOGFILE"

# Verify identical output
diff <(bash slow_stats.sh "$LOGFILE") <(fast_stats "$LOGFILE") \
  && echo "Outputs match" || echo "MISMATCH"

# Benchmark
TIMEFORMAT='%R'
printf 'slow_stats: '; time { bash slow_stats.sh "$LOGFILE" >/dev/null; }
printf 'fast_stats: '; time { fast_stats "$LOGFILE" >/dev/null; }
```

Chapter 7 — Security: Writing Safe Scripts

Shell scripts are glue code that runs close to the operating system, often with elevated privileges, and routinely handles data from untrusted sources — user input, environment variables, filenames, network responses. A single unquoted variable or careless `eval` can turn a maintenance script into a privilege-escalation vector. This chapter covers the specific vulnerabilities that affect Bash scripts and the patterns that reliably prevent them.

1 — Injection Vulnerabilities

Shell injection happens when attacker-controlled data is interpreted as shell syntax rather than plain text. The root cause is almost always unquoted variable expansion or dynamic command construction.

Argument injection via unquoted variables

```
# VULNERABLE: unquoted $filename undergoes word-splitting and globbing
filename="report 2024.pdf"
rm $filename        # runs: rm report 2024.pdf — two args, not one

filename="-rf /"
rm $filename        # runs: rm -rf /  ← catastrophic

# SAFE: always double-quote variable expansions
rm "$filename"

# SAFE: use -- to end option processing before filenames
rm -- "$filename"    # even "-rf /" is treated as a literal filename
```

Command injection via eval and dynamic commands

```
# VULNERABLE: user input passed to eval
read -r user_input
eval "echo Hello, $user_input"
```

```
# Input: "; rm -rf ~" → runs: echo Hello, ; rm -rf ~

# VULNERABLE: constructing commands by concatenation
cmd="convert $input_file -resize 800x $output_file"
eval "$cmd"
# input_file="x.png -write /etc/passwd x.png" → writes /etc/passwd

# SAFE: pass arguments as separate words — never concatenate into one string
convert "$input_file" -resize 800x "$output_file"

# SAFE: when eval is truly needed, quote with printf %q
eval "myvar=$(printf '%q' "$user_input")"
# printf %q shell-escapes the value so it can only ever be a string literal
```

Injection via $IFS manipulation

```
# If an attacker can set IFS, word-splitting changes everywhere
IFS="/"
path="/usr/bin/bash"
for part in $path; do echo "$part"; done   # splits on / not space

# Defence: reset IFS at the top of every script that may run in a
# manipulated environment, or always quote expansions (quotes suppress IFS)
IFS=$' \t\n'   # restore default at script start
```

> **The golden rule:** every variable that originates outside your script (arguments, environment, files, network, user input) is untrusted. Quote it, validate it, or both — never interpolate it raw into a command string.

2 — Safe Input Validation

```
# Allowlist approach — only accept known-good patterns
validate_username() {
  local name="$1"
  if [[ "$name" =~ ^[a-zA-Z0-9_-]{1,32}$ ]]; then
    return 0
```

```
  else
    printf 'Invalid username: %q\n' "$name" >&2
    return 1
  fi
}

validate_integer() {
  local n="$1"
  local min="${2:--2147483648}"
  local max="${3:-2147483647}"
  [[ "$n" =~ ^-?[0-9]+$ ]] || { echo "not an integer: $n" >&2; return 1; }
  (( n >= min && n <= max )) || { echo "out of range: $n" >&2; return 1; }
}

validate_path() {
  # Reject path traversal and null bytes
  local p="$1"
  local base="${2:-/safe/basedir}"   # required prefix
  [[ "$p" != *".."*  ]] || { echo "traversal attempt: $p" >&2; return 1; }
  [[ "$p" != *$'\0'* ]] || { echo "null byte in path" >&2; return 1; }
  # Resolve to canonical path then check prefix
  local real
  real=$(realpath -m "$p")   # -m: don't require the path to exist
  [[ "$real" == "$base"/* || "$real" == "$base" ]] || {
    echo "path escapes base: $real" >&2; return 1
  }
}

# Always validate before use
validate_username "$1" || exit 1
validate_path "$2" /var/uploads || exit 1
```

3 — Safe eval Patterns

`eval` is dangerous but sometimes unavoidable — for example when dynamically constructing variable names or sourcing configuration that must be Bash syntax. These patterns make it as safe as possible.

```
# printf %q — shell-quote a value so it survives one round of eval
declare -A config
```

```bash
safe_set_var() {
  local varname="$1"
  local value="$2"
  # Validate that varname is a legal identifier
  [[ "$varname" =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]] || {
    echo "invalid variable name: $varname" >&2; return 1
  }
  # printf %q escapes the value; eval sees:  varname='safe value'
  eval "$varname=$(printf '%q' "$value")"
}

safe_set_var greeting "hello; rm -rf ~"
echo "$greeting"   # hello; rm -rf ~  — stored as literal string, not executed

# declare -p round-trip is safe eval (output is already shell-quoted)
arr=( one two three )
serialised=$(declare -p arr)
eval "$serialised"   # safe: declare -p output is quoted by bash itself

# Sourcing config files — risks and mitigations
# Risk: source /user/provided/path.conf runs arbitrary code
# Mitigation 1: check ownership and permissions first
safe_source() {
  local file="$1"
  [[ -f "$file" ]]         || { echo "not a file: $file" >&2; return 1; }
  [[ ! -L "$file" ]]       || { echo "symlink refused: $file" >&2; return 1; }
  # Owned by root or current user, not world-writable
  local owner perms
  read -r perms _ owner _ <<< $(ls -la "$file")
  [[ "$owner" == "root" || "$owner" == "$USER" ]] || {
    echo "refusing to source file owned by $owner" >&2; return 1
  }
  [[ "${perms: -1}" != "w" ]] || {
    echo "world-writable config refused" >&2; return 1
  }
  source "$file"
}
```

4 — Temporary Files: Doing It Right

```
# VULNERABLE: predictable temp filename → symlink attack
tmpfile=/tmp/myapp_$$          # attacker creates symlink before we do
echo "secret data" > "$tmpfile"  # writes to attacker's target

# SAFE: mktemp creates a unique file atomically
tmpfile=$(mktemp)                      # /tmp/tmp.XXXXXXXXXX
tmpdir=$(mktemp -d)                    # directory
tmpfile=$(mktemp -t myapp.XXXXXXXXXX)    # custom prefix/template

# ALWAYS register cleanup in a trap so temp files are deleted on exit,
# error, or signal — including SIGINT
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

# Stacking traps — preserve existing EXIT handler
add_exit_trap() {
  local existing
  existing=$(trap -p EXIT | sed "s/trap -- '//;s/' EXIT//")
  trap "${existing:+${existing};} $1" EXIT
}
add_exit_trap 'rm -f "$tmpfile"'
add_exit_trap 'rm -rf "$tmpdir"'

# Use a private temp directory under $TMPDIR (respects user's temp preference)
WORK=$(mktemp -d "${TMPDIR:-/tmp}/myapp.XXXXXXXXXX")
trap 'rm -rf "$WORK"' EXIT

# Create files inside the private dir — no race risk from /tmp
secrets="${WORK}/secrets"
touch "$secrets"
chmod 600 "$secrets"
printf 'API_KEY=%s\n' "$API_KEY" > "$secrets"
```

5 — Privilege Separation and sudo

```
# Principle of least privilege: drop privileges as soon as possible

# Check if running as root — refuse if not expected
```

```bash
if (( EUID == 0 )); then
  echo "Do not run this script as root" >&2; exit 1
fi

# Or require root
if (( EUID != 0 )); then
  exec sudo -- "$0" "$@"   # re-exec ourselves under sudo
fi

# Privilege separation: do the privileged part in a minimal subshell,
# then drop back to the original user for the rest
run_as_root() {
  local user="${SUDO_USER:-$USER}"
  # Only allow specific whitelisted commands
  case "$1" in
    reload-nginx)  systemctl reload nginx ;;
    clear-cache)   find /var/cache/app -delete ;;
    *) echo "unknown privileged action: $1" >&2; return 1 ;;
  esac
}

# sudoers entry for script-specific commands (add via visudo):
# deploy ALL=(root) NOPASSWD: /usr/local/sbin/deploy-helper reload-nginx
# deploy ALL=(root) NOPASSWD: /usr/local/sbin/deploy-helper clear-cache

# NEVER pass user-supplied strings to sudo — validate first
# BAD:
#   sudo systemctl "$user_action" "$user_service"
# GOOD:
case "$user_action" in
  start|stop|restart|status) : ;;
  *) echo "invalid action" >&2; exit 1 ;;
esac
case "$user_service" in
  nginx|redis|postgres) : ;;
  *) echo "invalid service" >&2; exit 1 ;;
esac
sudo systemctl "$user_action" "$user_service"
```

6 — Secrets Handling

Credentials in scripts are a perennial source of breaches. The most common mistakes are storing secrets in environment variables, command-line arguments, or hardcoded in the script itself.

Why environment variables are risky

```
# Anyone who can read /proc/PID/environ sees your variables
# ps aux shows command-line arguments — including passwords passed as args

# BAD: hardcoded secret
DB_PASS="hunter2"

# BAD: secret in command-line argument
mysql -u root -phunter2   # visible in ps aux output

# BETTER: read from a file with restricted permissions
DB_PASS_FILE=/etc/myapp/db.secret   # chmod 400, owned by service user
DB_PASS=$(< "$DB_PASS_FILE")

# BETTER: use mysql option file instead of -p flag
# ~/.my.cnf or --defaults-extra-file=/path/to/file
mysql --defaults-extra-file=/etc/myapp/mysql.cnf -u myapp

# BEST: use a secrets manager and only hold in memory briefly
DB_PASS=$(vault kv get -field=password secret/myapp/db)
# Use DB_PASS immediately, then unset it
mysql -u myapp -p"$DB_PASS" -e "SELECT 1"
unset DB_PASS   # remove from environment as soon as possible
```

Passing secrets to child processes without the environment

```
# Use process substitution or a temp file to avoid the environment entirely

# Pass a secret via stdin instead of an argument
printf '%s\n' "$DB_PASS" | some-program --password-stdin

# Many tools support reading secrets from a file descriptor
```

```
exec 3<<<"$DB_PASS"
some-program --password-fd=3
exec 3<&-

# Scrubbing sensitive variables from the environment before exec'ing untrusted
code
unset AWS_SECRET_ACCESS_KEY DB_PASS GITHUB_TOKEN
# Or use env -i to start with a clean environment
env -i HOME="$HOME" PATH="$PATH" untrusted-program
```

Preventing secrets from appearing in logs and traces

```
# set -x traces every command — including ones with secrets in variables

# Suppress tracing for a sensitive block
set +x
PASSWORD=$(< /run/secrets/db_password)
set -x   # re-enable if needed

# Redirect xtrace away from the sensitive section
{
  local old_fd=$BASH_XTRACEFD
  exec 20>/dev/null; BASH_XTRACEFD=20
  SENSITIVE=$(read_secret)
  exec 20>&-; BASH_XTRACEFD="$old_fd"
}
```

7 — Hardening Script Startup

```
#!/usr/bin/env bash
# Recommended safety header for any script that handles untrusted data
# or runs with elevated privileges

set -euo pipefail
# -e  exit on first error
# -u  treat unset variables as errors (prevents silent empty expansions)
# -o pipefail  pipeline fails if any stage fails (not just the last)
```

```bash
# Restrict PATH to known-good locations — prevents PATH hijacking
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
export PATH

# Reset IFS to the default — env may have tampered with it
IFS=$' \t\n'

# Restrict file creation permissions (no group/world write on new files)
umask 077

# Prevent core dumps (may contain secrets)
ulimit -c 0

# Clear dangerous environment variables
unset CDPATH        # can redirect cd to unexpected directories
unset BASH_ENV      # sourced by bash when invoked as sh
unset ENV           # POSIX equivalent of BASH_ENV
unset BASH_XTRACEFD # don't inherit someone else's trace fd

# Absolute path for the script itself (guards against PATH tricks)
SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
```

8 — Common Vulnerability Checklist

Vulnerability	Trigger	Fix
Argument injection	Unquoted `$var` in command	Always quote: `"$var"`
Option injection	`cmd $var` where var starts with `-`	Use `cmd -- "$var"`
Command injection via eval	`eval "cmd $userdata"`	Never eval user data; use `printf %q`
PATH hijacking	Script inherits user's PATH	Set explicit PATH at startup
IFS manipulation	Inherited IFS changes word splitting	Reset IFS at script start

Symlink attack on temp files	Predictable `/tmp/name`	Use `mktemp`; trap EXIT for cleanup
Path traversal	User-supplied path with `..`	Validate with `realpath` and prefix check
Credential exposure	Secret in env var, CLI arg, or code	Use secret files or a secrets manager; `unset` after use
World-writable output	Default umask	`umask 077` at startup
Unchecked exit codes	Continuing after a failed command	`set -euo pipefail`
Unset variable expansion	`rm -rf "$dir/"` when `$dir` is empty	`set -u`; use `${var:?error}`
Glob in filenames	Untrusted filename containing `*?[]`	Quote all filenames; use `--`
TOCTOU (check-then-use)	Testing file, then acting — race window	Use atomic operations; `noclobber`, `flock`

Exercises

Exercise 1 — Vulnerability hunt

The following script contains at least six distinct security vulnerabilities. Identify each one, name the vulnerability class, and provide the corrected line. Then rewrite the complete function safely.

```
deploy_release() {
  version=$1
  deploy_dir="/var/www/$version"
  archive="/tmp/$version.tar.gz"
  curl -so $archive "https://releases.example.com/$version.tar.gz"
  tar -xzf $archive -C $deploy_dir
  config=$(cat /etc/myapp/deploy.conf)
  eval "$config"
  chmod -R 777 "$deploy_dir"
  echo "Deployed $version with DB_PASS=$DB_PASS"
}
```

```bash
# Vulnerabilities:
# 1. $1 not quoted → word splitting/glob if version contains spaces or sp
ecial chars
# 2. $archive unquoted in curl → argument injection (also predictable /tm
p path)
# 3. $archive and $deploy_dir unquoted in tar → argument injection
# 4. eval "$config" → arbitrary code execution from config file
# 5. chmod -R 777 → world-writable deployed files
# 6. echo "... DB_PASS=$DB_PASS" → credential leakage in logs/terminal

deploy_release() {
  local version="${1:?version required}"

  # 1. Validate version is a safe identifier (no path traversal, no injec
tion)
  [[ "$version" =~ ^[a-zA-Z0-9._-]+$ ]] || {
    printf 'Invalid version: %q\n' "$version" >&2; return 1
  }

  # 2. Use mktemp for the archive — not a predictable /tmp path
  local archive
  archive=$(mktemp -t deploy.XXXXXXXXXX.tar.gz)
  trap 'rm -f "$archive"' RETURN

  # 3. Quote all variables used in commands
  local deploy_dir="/var/www/$version"
  # Validate deploy_dir stays within /var/www/
  local real_dir
  real_dir=$(realpath -m "$deploy_dir")
  [[ "$real_dir" == "/var/www/"* ]] || {
    echo "deploy_dir escapes /var/www/" >&2; return 1
  }

  curl -so "$archive" "https://releases.example.com/$version.tar.gz"
  tar -xzf "$archive" -C "$deploy_dir"

  # 4. Source config only after safety checks — never eval raw file conte
nt
  safe_source /etc/myapp/deploy.conf
```

```
  # 5. Least-privilege permissions
  chmod -R 755 "$deploy_dir"
  find "$deploy_dir" -name '*.conf' -exec chmod 640 {} +

  # 6. Never log credentials
  printf 'Deployed %s successfully\n' "$version"
}
```

Exercise 2 — Safe argument parser

Write a function `safe_cli_parser` that processes these arguments: `--user NAME`, `--days N`, `--output FILE`, and an optional `--verbose` flag. Enforce:

- `NAME` matches `^[a-zA-Z0-9_-]{1,32}$`
- `N` is an integer between 1 and 365
- `FILE` resolves inside `/var/reports/` — reject traversal
- Unknown arguments cause an error with a usage message
- All three required arguments must be provided

Demonstrate it correctly rejecting: `--user "admin; rm -rf /"`, `--days 999`, and `--output "../../etc/passwd"`.

```
safe_cli_parser() {
  local user="" days="" output="" verbose=0

  while (( $# > 0 )); do
    case "$1" in
      --user)
        [[ $# -ge 2 ]] || { echo "--user requires a value" >&2; return 1;
}
        [[ "$2" =~ ^[a-zA-Z0-9_-]{1,32}$ ]] || {
          printf 'Invalid --user value: %q\n' "$2" >&2; return 1
        }
        user="$2"; shift 2 ;;
      --days)
        [[ $# -ge 2 ]] || { echo "--days requires a value" >&2; return 1;
}
```

```
        [[ "$2" =~ ^[0-9]+$ ]] && (( 10#$2 >= 1 && 10#$2 <= 365 )) || {
          printf '--days must be 1-365, got: %q\n' "$2" >&2; return 1
        }
        days="$2"; shift 2 ;;
      --output)
        [[ $# -ge 2 ]] || { echo "--output requires a value" >&2; return
1; }
        local real_out
        real_out=$(realpath -m "$2")
        [[ "$real_out" == "/var/reports/"* ]] || {
          printf '--output must be inside /var/reports/, got: %q\n' \
            "$real_out" >&2; return 1
        }
        output="$real_out"; shift 2 ;;
      --verbose) verbose=1; shift ;;
      *)
        printf 'Unknown argument: %q\n' "$1" >&2
        printf 'Usage: cmd --user NAME --days N --output FILE [--verbose]
\n' >&2
        return 1 ;;
    esac
  done

  for req in user days output; do
    [[ -n "${!req}" ]] || {
      printf '--%s is required\n' "$req" >&2; return 1
    }
  done

  printf 'OK: user=%s days=%s output=%s verbose=%d\n' \
    "$user" "$days" "$output" "$verbose"
}

# Rejection tests
safe_cli_parser --user "admin; rm -rf /" --days 7   --output /var/report
s/r.txt
safe_cli_parser --user alice             --days 999 --output /var/repor
ts/r.txt
safe_cli_parser --user alice             --days 7   --output ../../etc/
passwd
# All three should print errors and return 1
```

```
safe_cli_parser --user alice              --days 30  --output /var/repor
ts/alice.csv --verbose
# Should print: OK: user=alice days=30 output=/var/reports/alice.csv verb
ose=1
```

Exercise 3 — Secure secret loader

Write a `load_secrets FILE` function that reads a `KEY=VALUE` config file and exports only the listed variables. Requirements:

- Reject the file if it is world-readable (`chmod o-r` check)
- Reject the file if it is a symlink
- Reject the file if it is not owned by the current user or root
- Parse lines manually — do *not* `source` or `eval` the file
- Accept only keys matching `^[A-Z_][A-Z0-9_]*$` and skip malformed lines with a warning
- Strip single and double quotes from values
- After loading, overwrite the variable storing the file path with an empty string and unset it

---

```
load_secrets() {
  local file="$1"

  # Existence
  [[ -f "$file" ]] || { printf 'load_secrets: not a file: %s\n' "$file" >
&2; return 1; }

  # Not a symlink
  [[ ! -L "$file" ]] || { printf 'load_secrets: symlinks refused: %s\n'
"$file" >&2; return 1; }

  # Permissions: not world-readable
  local mode owner
  read -r mode _ owner <<< $(stat -c '%a %U' "$file")
  (( 10#$mode % 10 == 0 )) || {
    printf 'load_secrets: file is world-readable (mode %s): %s\n' \
      "$mode" "$file" >&2; return 1
  }

  # Ownership: current user or root
```

```
  [[ "$owner" == "$USER" || "$owner" == "root" ]] || {
    printf 'load_secrets: file owned by %s, expected %s or root\n' \
      "$owner" "$USER" >&2; return 1
  }

  # Parse manually — no eval/source
  local lineno=0 key value raw
  while IFS= read -r raw || [[ -n "$raw" ]]; do
    (( lineno++ ))
    # Skip blank lines and comments
    [[ $raw =~ ^[[:space:]]*(#|$) ]] && continue
    # Must be KEY=VALUE
    [[ $raw =~ ^([A-Z_][A-Z0-9_]*)=(.*) ]] || {
      printf 'load_secrets: skipping malformed line %d: %q\n' \
        "$lineno" "$raw" >&2; continue
    }
    key="${BASH_REMATCH[1]}"
    value="${BASH_REMATCH[2]}"
    # Strip surrounding single or double quotes
    value="${value#\"}"; value="${value%\"}"
    value="${value#\'}"; value="${value%\'}"
    # Export without eval
    export "$key"="$value"
  done < "$file"

  # Scrub the file path variable
  file=""
  unset file
}

# Test
touch /tmp/test_secrets.env
chmod 600 /tmp/test_secrets.env
printf 'DB_PASS="s3cr3t"\nAPI_KEY=abc123\nBAD LINE\n# comment\n' \
  > /tmp/test_secrets.env
load_secrets /tmp/test_secrets.env
echo "DB_PASS=$DB_PASS API_KEY=$API_KEY"
rm /tmp/test_secrets.env
```

Exercise 4 — Hardened script template

Write a complete, production-ready Bash script skeleton called `hardened_template.sh` that incorporates every defence from this chapter. It should:

- Set all recommended startup options (`set -euo pipefail`, fixed PATH, IFS, umask, ulimit)
- Unset dangerous environment variables
- Declare a `die CODE MESSAGE` helper and a `cleanup` EXIT trap
- Validate at least one argument using an allowlist regex
- Create a private temp directory via `mktemp -d` registered in the EXIT trap
- Demonstrate safe secret loading (read from a file, never echo)
- Include a `--dry-run` flag that suppresses all destructive operations
- Be fully ShellCheck-clean (no SC warnings)

```
#!/usr/bin/env bash
# hardened_template.sh — production-ready secure script skeleton
set -euo pipefail

# ── Security hardening ──────────────────────────────────────
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
export PATH
IFS=$' \t\n'
umask 077
ulimit -c 0
unset CDPATH BASH_ENV ENV

# ── Script identity ─────────────────────────────────────────
SCRIPT_DIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
SCRIPT_NAME="${0##*/}"

# ── Globals ─────────────────────────────────────────────────
DRY_RUN=0
WORK=""

# ── Helpers ─────────────────────────────────────────────────
die() {
  local code="$1"; shift
  printf '%s: ERROR: %s\n' "$SCRIPT_NAME" "$*" >&2
  exit "$code"
```

```
}

log() { printf '[%(%Y-%m-%d %H:%M:%S)T] %s\n' -1 "$*"; }

cleanup() {
  [[ -n "$WORK" ]] && rm -rf "$WORK"
}
trap cleanup EXIT

# ── Argument parsing ──────────────────────────────────────────
usage() {
  printf 'Usage: %s [--dry-run] --env (prod|staging|dev)\n' "$SCRIPT_NAM
E" >&2
  exit 1
}

TARGET_ENV=""
while (( $# > 0 )); do
  case "$1" in
    --dry-run)  DRY_RUN=1; shift ;;
    --env)
      [[ $# -ge 2 ]] || die 1 "--env requires a value"
      [[ "$2" =~ ^(prod|staging|dev)$ ]] ||
        die 1 "--env must be prod, staging, or dev; got: $(printf '%q'
"$2")"
      TARGET_ENV="$2"; shift 2 ;;
    -h|--help) usage ;;
    *) die 1 "Unknown argument: $(printf '%q' "$1")" ;;
  esac
done
[[ -n "$TARGET_ENV" ]] || die 1 "--env is required"

# ── Setup ─────────────────────────────────────────────────────
WORK=$(mktemp -d "${TMPDIR:-/tmp}/${SCRIPT_NAME%.sh}.XXXXXXXXXX")
log "Work directory: $WORK"

# ── Secret loading (no eval, no echo) ─────────────────────────
SECRET_FILE="${SCRIPT_DIR}/secrets/${TARGET_ENV}.env"
if [[ -f "$SECRET_FILE" ]]; then
  while IFS='=' read -r k v || [[ -n "$k" ]]; do
    [[ $k =~ ^[A-Z_][A-Z0-9_]*$ ]] || continue
```

```
    v="${v#\"}"; v="${v%\"}"
    export "$k"="$v"
  done < "$SECRET_FILE"
fi

# ── Main work ──────────────────────────────────────────────────
log "Starting deployment to $TARGET_ENV"
if (( DRY_RUN )); then
  log "[DRY RUN] would deploy to $TARGET_ENV — skipping destructive step
s"
else
  log "Deploying..."
  # ... real work here ...
fi

log "Done"
```

Chapter 8 — Advanced Testing with BATS

BATS (Bash Automated Testing System) turns shell scripts into first-class testable software. Where ad-hoc `if` checks give you fragile one-off verification, BATS gives you structured test suites with TAP output, fixtures, mocking, and CI integration. This chapter covers the full depth of BATS: test organisation, lifecycle hooks, filesystem fixtures, command mocking, testing scripts that modify the system, and wiring everything into a pipeline.

1 — BATS Architecture and Installation

Modern BATS is split across three repositories. You need all three for a complete setup.

Package	Provides
`bats-core`	The test runner, `@test` syntax, TAP output
`bats-support`	Helper functions: `fail`, `assert` output formatting
`bats-assert`	Assertion library: `assert_output`, `assert_success`, etc.
`bats-file`	Filesystem assertions: `assert_file_exists`, `assert_dir_empty`, etc.

```
# Install via git submodules (recommended for project-local install)
git submodule add https://github.com/bats-core/bats-core    test/bats
git submodule add https://github.com/bats-core/bats-support  test/test_helper/
bats-support
git submodule add https://github.com/bats-core/bats-assert   test/test_helper/
bats-assert
git submodule add https://github.com/bats-core/bats-file     test/test_helper/
bats-file

# Or via npm (if Node is available)
npm install --save-dev bats

# Or via Homebrew / apt
brew install bats-core
apt-get install bats
```

```
# Run a test file
bats test/my_test.bats

# Run all .bats files recursively
bats --recursive test/

# Parallel execution — run test files in parallel
bats --jobs 4 --recursive test/

# Output formats
bats --formatter tap        test/   # TAP (default)
bats --formatter pretty     test/   # coloured human output
bats --formatter junit      test/   # JUnit XML for CI systems
bats --report-formatter junit --output reports/ test/
```

2 — Test File Structure

```
#!/usr/bin/env bats
# test/mylib.bats

# Load helper libraries
setup_file() {
  # Runs ONCE before the first test in this file
  load 'test_helper/bats-support/load'
  load 'test_helper/bats-assert/load'
  load 'test_helper/bats-file/load'

  # Make the script under test available
  export SCRIPT="${BATS_TEST_DIRNAME}/../bin/myapp.sh"
  # BATS_TEST_DIRNAME is the directory containing this .bats file
}

teardown_file() {
  # Runs ONCE after the last test in this file
  # Good for: removing shared DB state, stopping a server, etc.
  :
}
```

```
setup() {
  # Runs before EVERY individual test
  # BATS_TEST_TMPDIR is a per-test temp directory (auto-cleaned)
  TEST_HOME="${BATS_TEST_TMPDIR}/home"
  mkdir -p "$TEST_HOME"
}

teardown() {
  # Runs after EVERY individual test — even if the test failed
  # BATS_TEST_TMPDIR is automatically removed after teardown
  :
}

# A basic test
@test "myapp exits 0 with valid input" {
  run "$SCRIPT" --input "hello"
  assert_success
}

@test "myapp prints expected output" {
  run "$SCRIPT" --input "hello"
  assert_output "HELLO"
}

@test "myapp exits 1 with empty input" {
  run "$SCRIPT" --input ""
  assert_failure
  assert_output --partial "input required"
}
```

3 — The `run` Command and Assertions

The `run` command executes a command and captures its output and exit status without causing the test to fail. The results land in three variables: `$status`, `$output`, and `$lines` (array).

```
@test "run captures status and output" {
  run printf '%s\n%s\n' "line one" "line two"
```

```
  # $status — numeric exit code
  [ "$status" -eq 0 ]

  # $output — full stdout as a single string
  [ "$output" = "line one
line two" ]

  # $lines — indexed array of output lines
  [ "${lines[0]}" = "line one" ]
  [ "${lines[1]}" = "line two" ]
}

# run --separate-stderr (bats-core 1.5+): captures stdout and stderr separatel
y
@test "stderr is separate from stdout" {
  run --separate-stderr bash -c 'echo out; echo err >&2'
  assert_output "out"
  assert_stderr "err"
}

# bats-assert assertions
@test "assert_output variants" {
  run echo "The quick brown fox"
  assert_output                        # non-empty
  assert_output "The quick brown fox"    # exact match
  assert_output --partial "brown"        # substring
  assert_output --regexp  'quick .* fox' # ERE regex
  refute_output --partial "cat"          # must NOT contain
}

@test "assert_line checks individual lines" {
  run printf 'alpha\nbeta\ngamma\n'
  assert_line "alpha"                    # any line equals "alpha"
  assert_line --index 1 "beta"           # line[1] == "beta"
  assert_line --partial "amm"            # any line contains "amm"
  refute_line "delta"                    # "delta" must not appear
}

# assert_equal, assert_not_equal — direct value comparison
@test "assert_equal" {
  local result
```

```
  result=$(echo "hello" | tr 'a-z' 'A-Z')
  assert_equal "$result" "HELLO"
}
```

4 — Fixtures and Test Isolation

BATS provides per-test isolation through `$BATS_TEST_TMPDIR` — a unique temporary directory created before each test and deleted after its teardown. Build all test filesystem state inside it.

```
setup() {
  load 'test_helper/bats-support/load'
  load 'test_helper/bats-assert/load'
  load 'test_helper/bats-file/load'

  # Build a reproducible directory structure for each test
  FIXTURE_DIR="${BATS_TEST_TMPDIR}/fixture"
  mkdir -p "$FIXTURE_DIR"/{src,dest,logs}

  # Populate with known-content files
  printf 'line1\nline2\nline3\n' > "$FIXTURE_DIR/src/input.txt"
  printf 'ERROR: disk full\nINFO: started\nWARN: retry\n' \
    > "$FIXTURE_DIR/logs/app.log"
}

@test "script copies file to dest" {
  run ../bin/copy_files.sh "$FIXTURE_DIR/src" "$FIXTURE_DIR/dest"
  assert_success
  assert_file_exists "$FIXTURE_DIR/dest/input.txt"
}

@test "script leaves source intact" {
  run ../bin/copy_files.sh "$FIXTURE_DIR/src" "$FIXTURE_DIR/dest"
  assert_file_exists     "$FIXTURE_DIR/src/input.txt"
  assert_file_not_empty  "$FIXTURE_DIR/src/input.txt"
}

@test "log parser counts errors" {
  run ../bin/parse_log.sh "$FIXTURE_DIR/logs/app.log"
  assert_success
```

```
  assert_output --partial "errors: 1"
}
```

Static fixture files

```
# Store static fixture files alongside your tests
# Conventional layout:
#   test/
#   ├── fixtures/
#   │   ├── config_valid.conf
#   │   ├── config_malformed.conf
#   │   └── sample_data.csv
#   └── myapp_test.bats

setup() {
  # BATS_TEST_DIRNAME always points to the directory of the .bats file
  FIXTURES="${BATS_TEST_DIRNAME}/fixtures"
}

@test "accepts valid config" {
  run ../bin/myapp.sh --config "$FIXTURES/config_valid.conf"
  assert_success
}

@test "rejects malformed config" {
  run ../bin/myapp.sh --config "$FIXTURES/config_malformed.conf"
  assert_failure
  assert_output --partial "invalid config"
}
```

5 — Mocking Commands with PATH Manipulation

The cleanest way to mock an external command in BATS is to create a fake executable on a PATH that is prepended for the duration of the test. The real command is never called; your mock controls what happens.

```
setup() {
  load 'test_helper/bats-support/load'
  load 'test_helper/bats-assert/load'

  # Create a directory for mock executables, prepend to PATH
  MOCK_DIR="${BATS_TEST_TMPDIR}/mocks"
  mkdir -p "$MOCK_DIR"
  export PATH="$MOCK_DIR:$PATH"
}

# Helper: create a mock that always succeeds and records its arguments
make_mock() {
  local name="$1"; shift
  local script="${MOCK_DIR}/${name}"
  printf '#!/usr/bin/env bash\nprintf "%%s\n" "$*" >> "%s/%s.calls"\n%s\n' \
    "$MOCK_DIR" "$name" "${*:-exit 0}" > "$script"
  chmod +x "$script"
}

# Helper: read recorded calls
mock_calls() {
  cat "${MOCK_DIR}/${1}.calls" 2>/dev/null
}

# Helper: count how many times a command was called
mock_call_count() {
  wc -l < "${MOCK_DIR}/${1}.calls" 2>/dev/null || echo 0
}

@test "deploy script calls systemctl reload" {
  make_mock systemctl
  make_mock rsync

  run ../bin/deploy.sh staging

  assert_success
  # Verify rsync was called exactly once
  assert_equal "$(mock_call_count rsync)" "1"
  # Verify systemctl was called with "reload nginx"
  run mock_calls systemctl
  assert_output --partial "reload nginx"
```

```
}

@test "deploy script handles rsync failure" {
  # Mock rsync to fail
  make_mock rsync 'exit 1'
  make_mock systemctl

  run ../bin/deploy.sh staging
  assert_failure
  # systemctl should NOT have been called after rsync failure
  assert_equal "$(mock_call_count systemctl)" "0"
}
```

Mocking with output

```
# Mock that prints specific output (useful for testing command substitutions)
make_mock_output() {
  local name="$1"
  local output="$2"
  local exit_code="${3:-0}"
  printf '#!/usr/bin/env bash\nprintf "%%s\n" %s\nexit %d\n' \
    "$(printf '%q' "$output")" "$exit_code" > "${MOCK_DIR}/${name}"
  chmod +x "${MOCK_DIR}/${name}"
}

@test "script uses hostname in output" {
  make_mock_output hostname "testserver-01"

  run ../bin/status.sh
  assert_output --partial "testserver-01"
}

@test "script handles curl failure gracefully" {
  make_mock_output curl "Connection refused" 7   # exit 7 = curl connect fail

  run ../bin/health_check.sh
  assert_failure
  assert_output --partial "health check failed"
}
```

6 — Testing Scripts That Modify the Filesystem

```
setup() {
  load 'test_helper/bats-support/load'
  load 'test_helper/bats-assert/load'
  load 'test_helper/bats-file/load'

  # Every test gets its own isolated filesystem
  FS_ROOT="${BATS_TEST_TMPDIR}/fs"
  mkdir -p "$FS_ROOT"/{etc,var/log,home/alice,backup}

  # Point the script's config at our fake root
  export APP_ROOT="$FS_ROOT"
  export CONFIG_DIR="$FS_ROOT/etc"
  export LOG_DIR="$FS_ROOT/var/log"
}

@test "install creates config file with correct permissions" {
  run ../bin/install.sh
  assert_success
  assert_file_exists       "$FS_ROOT/etc/myapp.conf"
  assert_file_permissions  "$FS_ROOT/etc/myapp.conf" 600
  assert_file_owner        "$FS_ROOT/etc/myapp.conf" "$USER"
}

@test "backup script creates archive in backup dir" {
  # Seed data to back up
  printf 'important data\n' > "$FS_ROOT/var/log/app.log"

  run ../bin/backup.sh "$FS_ROOT/var/log" "$FS_ROOT/backup"
  assert_success
  assert_dir_not_empty "$FS_ROOT/backup"
}

@test "cleanup script removes files older than 30 days" {
  # Create a file with a known old timestamp
  local old_file="$FS_ROOT/var/log/old.log"
  local new_file="$FS_ROOT/var/log/new.log"
  touch "$old_file" "$new_file"
  # Back-date old_file by 31 days using touch -d
```

```
  touch -d '31 days ago' "$old_file"

  run ../bin/cleanup.sh "$FS_ROOT/var/log" --days 30
  assert_success
  assert_file_not_exist  "$old_file"
  assert_file_exists     "$new_file"
}

@test "cleanup in dry-run mode does not delete anything" {
  local old_file="$FS_ROOT/var/log/old.log"
  touch "$old_file"
  touch -d '31 days ago' "$old_file"

  run ../bin/cleanup.sh "$FS_ROOT/var/log" --days 30 --dry-run
  assert_success
  assert_file_exists  "$old_file"   # must still be there
  assert_output --partial "would delete"
}
```

7 — Advanced Mocking Techniques

Mocking functions within a sourced library

```
# When testing a library (not a script), source it and override functions
setup() {
  load 'test_helper/bats-support/load'
  load 'test_helper/bats-assert/load'
  # Source the library under test
  source "${BATS_TEST_DIRNAME}/../lib/deploy_lib.sh"
}

@test "notify_slack is called on success" {
  # Override the real notify_slack with a mock in the same shell
  SLACK_CALLED=0
  notify_slack() { (( SLACK_CALLED++ )) || true; }

  # Call the function under test (which internally calls notify_slack)
  run deploy_app staging
```

```
  assert_success
  assert_equal "$SLACK_CALLED" "1"
}
```

Stateful mocks — returning different values on successive calls

```
@test "retry logic calls command up to 3 times" {
  # First two calls fail, third succeeds
  CALL_COUNT=0
  flaky_service() {
    (( CALL_COUNT++ )) || true
    (( CALL_COUNT < 3 )) && return 1   # fail first 2 times
    return 0
  }

  run retry_until_success 5 flaky_service
  assert_success
  assert_equal "$CALL_COUNT" "3"
}

# File-backed stateful mock (works for child-process mocks too)
setup() {
  MOCK_DIR="${BATS_TEST_TMPDIR}/mocks"
  mkdir -p "$MOCK_DIR"
  export PATH="$MOCK_DIR:$PATH"
  # Mock that fails twice then succeeds — uses a counter file
  printf '#!/usr/bin/env bash
COUNT_FILE="%s/ping.count"
count=$(cat "$COUNT_FILE" 2>/dev/null || echo 0)
echo $(( count + 1 )) > "$COUNT_FILE"
(( count >= 2 )) && exit 0 || exit 1
' "$MOCK_DIR" > "$MOCK_DIR/ping"
  chmod +x "$MOCK_DIR/ping"
}

@test "wait_for_host retries on ping failure" {
  run ../bin/wait_for_host.sh example.com
  assert_success
  # ping should have been called 3 times (2 failures + 1 success)
```

```
  assert_equal "$(< "$MOCK_DIR/ping.count")" "3"
}
```

8 — Testing Edge Cases and Error Paths

```
# Test that a script fails correctly — not just that it fails
@test "missing required argument prints usage" {
  run ../bin/myapp.sh   # no args
  assert_failure
  assert_output --partial "Usage:"
  assert_equal "$status" "1"
}

# Test specific exit codes
@test "file not found returns exit 2" {
  run ../bin/myapp.sh --input /nonexistent/file
  assert_equal "$status" "2"
}

# Test behaviour with unusual filenames (spaces, special chars)
@test "handles filename with spaces" {
  local f="$BATS_TEST_TMPDIR/file with spaces.txt"
  printf 'data\n' > "$f"
  run ../bin/process.sh "$f"
  assert_success
}

@test "handles empty file" {
  local f="$BATS_TEST_TMPDIR/empty.txt"
  touch "$f"
  run ../bin/process.sh "$f"
  assert_success
  assert_output --partial "0 lines processed"
}

# Skipping tests conditionally
@test "docker-based test" {
  skip_if_missing docker
```

```
  run ../bin/containerise.sh
  assert_success
}

# skip_if_missing helper
skip_if_missing() {
  command -v "$1" &>/dev/null || skip "$1 not available"
}

# skip built-in — marks test as skipped rather than failed
@test "root-only operation" {
  (( EUID == 0 )) || skip "requires root"
  run ../bin/set_caps.sh
  assert_success
}
```

9 — CI/CD Integration

```
# .github/workflows/test.yml
# ── GitHub Actions ──────────────────────────────────────────
```

```
name: Tests
on: [push, pull_request]
jobs:
  bats:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          submodules: recursive      # fetch bats-* submodules
      - name: Install bats
        run: sudo apt-get install -y bats
      - name: Run tests
        run: bats --formatter junit --output reports/ --recursive test/
      - name: Publish results
        uses: mikepenz/action-junit-report@v4
        if: always()
```

```
        with:
          report_paths: 'reports/*.xml'
```

```
# Makefile targets for local development
test:
        bats --recursive test/

test-watch:
        while true; do \
          inotifywait -qre close_write bin/ lib/ test/ 2>/dev/null; \
          clear; bats --formatter pretty --recursive test/; \
        done

test-coverage:
        # kcov is the closest thing to coverage for bash
        kcov --include-path=bin/,lib/ coverage/ \
          bats --recursive test/
```

Recommended project layout:

```
project/
├── bin/                ← scripts under test
├── lib/                ← sourced libraries under test
├── test/
│   ├── fixtures/       ← static input files
│   ├── test_helper/    ← bats-support, bats-assert, bats-file (submodul
es)
│   ├── unit/           ← tests for individual functions
│   │   └── lib_utils.bats
│   └── integration/    ← end-to-end script tests
│       └── deploy.bats
└── Makefile
```

Exercises

Exercise 1 — Test a string utility library

Given this library at `lib/strings.sh`:

```
str_trim()  { local s="$1"; s="${s#"${s%%[! $'\t']*}"}"; printf '%s'
"${s%"${s##*[! $'\t']}"}"; }
str_upper() { printf '%s' "${1^^}"; }
str_lower() { printf '%s' "${1,,}"; }
str_contains() { [[ "$1" == *"$2"* ]]; }
str_repeat() { printf "${1}%.0s" $(seq 1 "$2"); }
```

Write a complete BATS test file covering: normal cases for all five functions, edge cases (empty string, single char, string that IS the needle), and at least one failure case per function. Use `setup` to load the library and helpers. Include a test for a deliberately broken behaviour — pick one function to implement incorrectly, verify the test catches it, then fix it.

```
#!/usr/bin/env bats
# test/unit/strings_test.bats

setup() {
  load '../test_helper/bats-support/load'
  load '../test_helper/bats-assert/load'
  source "${BATS_TEST_DIRNAME}/../../lib/strings.sh"
}

# ── str_trim ──────────────────────────────────────────────────
@test "str_trim: removes leading spaces" {
  run str_trim "   hello"
  assert_output "hello"
}
@test "str_trim: removes trailing spaces" {
  run str_trim "hello   "
  assert_output "hello"
}
@test "str_trim: removes both ends" {
  run str_trim "  hello world  "
  assert_output "hello world"
}
@test "str_trim: empty string returns empty" {
  run str_trim ""
  assert_output ""
```

```
}
@test "str_trim: all whitespace returns empty" {
  run str_trim "     "
  assert_output ""
}

# ── str_upper / str_lower ──────────────────────────────────
@test "str_upper: lowercases to uppercase" {
  run str_upper "hello world"
  assert_output "HELLO WORLD"
}
@test "str_upper: already uppercase is unchanged" {
  run str_upper "HELLO"
  assert_output "HELLO"
}
@test "str_lower: uppercases to lowercase" {
  run str_lower "HELLO WORLD"
  assert_output "hello world"
}
@test "str_lower: mixed case" {
  run str_lower "HeLLo"
  assert_output "hello"
}

# ── str_contains ───────────────────────────────────────────
@test "str_contains: needle in middle" {
  run str_contains "hello world" "lo wo"
  assert_success
}
@test "str_contains: needle IS the string" {
  run str_contains "hello" "hello"
  assert_success
}
@test "str_contains: needle not present" {
  run str_contains "hello" "xyz"
  assert_failure
}
@test "str_contains: empty needle is always found" {
  run str_contains "hello" ""
  assert_success
}
```

```
# ── str_repeat ──────────────────────────────────────────────
@test "str_repeat: repeats 3 times" {
  run str_repeat "ab" 3
  assert_output "ababab"
}
@test "str_repeat: zero times returns empty" {
  run str_repeat "x" 0
  assert_output ""
}
@test "str_repeat: single char" {
  run str_repeat "-" 5
  assert_output "-----"
}
```

Exercise 2 — Mock-based deploy test

Write a BATS test file for the following `bin/deploy.sh` script. Use PATH-based mocks for `rsync`, `systemctl`, and `curl`. Cover: happy path (all succeed), rsync failure (systemctl not called), systemctl failure (curl still called for alert), and missing argument.

```
#!/usr/bin/env bash
set -euo pipefail
ENV="${1:?Usage: deploy.sh ENV}"
notify() { curl -s "$WEBHOOK_URL" -d "status=$1"; }
notify starting
rsync -a dist/ "deploy@${ENV}.example.com:/var/www/"
systemctl --host "$ENV.example.com" reload nginx
notify done
```

```
#!/usr/bin/env bats
# test/integration/deploy_test.bats

setup() {
  load '../test_helper/bats-support/load'
  load '../test_helper/bats-assert/load'
```

```
  MOCK_DIR="${BATS_TEST_TMPDIR}/mocks"
  mkdir -p "$MOCK_DIR"
  export PATH="$MOCK_DIR:$PATH"
  export WEBHOOK_URL="http://fake-webhook"
  SCRIPT="${BATS_TEST_DIRNAME}/../../bin/deploy.sh"
}

_make_mock() {
  local name="$1" exit_code="${2:-0}"
  local calls_file="$MOCK_DIR/$name.calls"
  printf '#!/usr/bin/env bash\nprintf "%%s\n" "$*" >> "%s"\nexit %d\n' \
    "$calls_file" "$exit_code" > "$MOCK_DIR/$name"
  chmod +x "$MOCK_DIR/$name"
}

_call_count() {
  wc -l < "$MOCK_DIR/$1.calls" 2>/dev/null || printf '0'
}

_calls() {
  cat "$MOCK_DIR/$1.calls" 2>/dev/null || true
}

# ── Tests ─────────────────────────────────────────────────────

@test "happy path: all commands called in order" {
  _make_mock rsync
  _make_mock systemctl
  _make_mock curl

  run bash "$SCRIPT" staging
  assert_success

  # curl called twice (starting + done)
  assert_equal "$(_call_count curl)"       "2"
  assert_equal "$(_call_count rsync)"      "1"
  assert_equal "$(_call_count systemctl)"  "1"

  # Verify notification content
  run _calls curl
  assert_line --partial "status=starting"
```

```
  assert_line --partial "status=done"
}

@test "rsync failure: systemctl not called" {
  _make_mock rsync 1   # rsync fails
  _make_mock systemctl
  _make_mock curl

  run bash "$SCRIPT" staging
  assert_failure
  assert_equal "$(_call_count systemctl)" "0"
}

@test "systemctl failure: curl still called for done/error" {
  _make_mock rsync
  _make_mock systemctl 1   # systemctl fails
  _make_mock curl

  # set -e means the script exits after systemctl fails —
  # 'done' notification is not sent; that is expected behaviour
  run bash "$SCRIPT" staging
  assert_failure
  # 'starting' notification should have been sent
  run _calls curl
  assert_line --partial "status=starting"
}

@test "missing argument exits 1 with usage message" {
  _make_mock rsync
  _make_mock systemctl
  _make_mock curl

  run bash "$SCRIPT"   # no ENV argument
  assert_failure
  assert_output --partial "Usage:"
  # No commands should have been called
  assert_equal "$(_call_count curl)"  "0"
  assert_equal "$(_call_count rsync)" "0"
}
```

Exercise 3 — Filesystem mutation test

Write a BATS test suite for a `bin/rotate_logs.sh LOGDIR MAX_DAYS` script that deletes files older than MAX_DAYS and compresses files older than half that age. Use `touch -d` to seed files with specific ages. Test: files younger than MAX_DAYS/2 are untouched; files between MAX_DAYS/2 and MAX_DAYS are compressed (renamed to `.gz`); files older than MAX_DAYS are deleted; the script refuses to operate on a path outside `/var/log` (when run against the real filesystem — mock this with an environment variable override).

```
#!/usr/bin/env bats
# test/integration/rotate_logs_test.bats

setup() {
  load '../test_helper/bats-support/load'
  load '../test_helper/bats-assert/load'
  load '../test_helper/bats-file/load'

  LOGDIR="${BATS_TEST_TMPDIR}/logs"
  mkdir -p "$LOGDIR"

  # Override the allowed base path for testing
  export LOG_BASE_PATH="$BATS_TEST_TMPDIR"

  MAX_DAYS=30

  # Seed test files at specific ages
  # young: 5 days old — should be untouched
  touch -d '5 days ago'  "$LOGDIR/young.log"
  # middle: 20 days old — older than MAX/2 (15), should be compressed
  touch -d '20 days ago' "$LOGDIR/middle.log"
  # old: 35 days old — older than MAX (30), should be deleted
  touch -d '35 days ago' "$LOGDIR/old.log"
  SCRIPT="${BATS_TEST_DIRNAME}/../../bin/rotate_logs.sh"
}

@test "young file is untouched" {
  run bash "$SCRIPT" "$LOGDIR" "$MAX_DAYS"
  assert_success
  assert_file_exists    "$LOGDIR/young.log"
```

```
  assert_file_not_exist "$LOGDIR/young.log.gz"
}

@test "middle-aged file is compressed" {
  run bash "$SCRIPT" "$LOGDIR" "$MAX_DAYS"
  assert_success
  assert_file_not_exist "$LOGDIR/middle.log"
  assert_file_exists    "$LOGDIR/middle.log.gz"
}

@test "old file is deleted" {
  run bash "$SCRIPT" "$LOGDIR" "$MAX_DAYS"
  assert_success
  assert_file_not_exist "$LOGDIR/old.log"
  assert_file_not_exist "$LOGDIR/old.log.gz"
}

@test "path outside LOG_BASE_PATH is rejected" {
  # Override to a strict base path that LOGDIR doesn't satisfy
  export LOG_BASE_PATH=/var/log
  run bash "$SCRIPT" "$LOGDIR" "$MAX_DAYS"
  assert_failure
  assert_output --partial "outside"
  # Verify no files were touched
  assert_file_exists "$LOGDIR/old.log"
}

@test "dry-run reports actions without modifying filesystem" {
  run bash "$SCRIPT" "$LOGDIR" "$MAX_DAYS" --dry-run
  assert_success
  assert_file_exists "$LOGDIR/old.log"
  assert_file_exists "$LOGDIR/middle.log"
  assert_output --partial "would delete"
  assert_output --partial "would compress"
}
```

Exercise 4 — CI configuration and Makefile

Write a complete `Makefile` and `.github/workflows/test.yml` for a project that has BATS tests. Requirements:

- `make test` — runs all tests with pretty output
- `make test-ci` — runs with JUnit XML output to `reports/`
- `make test-unit` and `make test-integration` — run subsets
- `make lint` — runs `shellcheck` on all `.sh` files
- `make test-watch` — re-runs tests on file change (using `inotifywait` or `fswatch`)
- The GitHub Actions workflow runs lint then tests, publishes JUnit results, and fails the workflow if any test fails

---

```
# Makefile
BATS       := test/bats/bin/bats
TEST_DIR   := test
REPORT_DIR := reports
SHELL_SRC  := $(shell find bin lib -name '*.sh' 2>/dev/null)

.PHONY: test test-ci test-unit test-integration lint test-watch clean

test:
	$(BATS) --formatter pretty --recursive $(TEST_DIR)/

test-ci: | $(REPORT_DIR)
	$(BATS) --formatter junit --output $(REPORT_DIR) \
	        --recursive $(TEST_DIR)/

test-unit:
	$(BATS) --formatter pretty --recursive $(TEST_DIR)/unit/

test-integration:
	$(BATS) --formatter pretty --recursive $(TEST_DIR)/integration/

lint:
	shellcheck $(SHELL_SRC)

test-watch:
	@if command -v inotifywait >/dev/null 2&1; then \
	  while true; do \
	    inotifywait -qre close_write bin/ lib/ $(TEST_DIR)/ 2/dev/nul
l; \
```

```
            clear; $(MAKE) test; \
          done; \
        elif command -v fswatch >/dev/null 2&1; then \
          fswatch -o bin/ lib/ $(TEST_DIR)/ | xargs -n1 sh -c 'clear; $(M
AKE) test'; \
        else \
          echo "Install inotify-tools (Linux) or fswatch (macOS)"; exit
1; \
        fi

$(REPORT_DIR):
        mkdir -p $(REPORT_DIR)

clean:
        rm -rf $(REPORT_DIR)
```

```yaml
# .github/workflows/test.yml
name: CI

on:
  push:
    branches: [main, 'feature/**']
  pull_request:

jobs:
  lint-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4
        with:
          submodules: recursive

      - name: Install dependencies
        run: |
          sudo apt-get update -q
          sudo apt-get install -y shellcheck

      - name: Lint
```

```
      run: make lint

    - name: Run tests
      run: make test-ci

    - name: Publish test results
      uses: mikepenz/action-junit-report@v4
      if: always()
      with:
        report_paths: 'reports/*.xml'
        fail_on_failure: true
```

Chapter 9 — Daemon Scripts and Process Management

A well-written daemon is invisible when healthy and unambiguous when broken. This chapter covers the full lifecycle: the double-fork daemonisation pattern, robust PID files, idiomatic start/stop/status dispatch, and native integration with systemd — the init system on every modern Linux distribution. You will also see how to handle signals cleanly, manage child processes, and write the `.service` unit that makes all of the hand-crafted shell daemonisation unnecessary in a systemd world.

1 — Why Daemonisation Is Complicated

A daemon is a process that runs detached from any controlling terminal, with its file descriptors closed, in its own session and process group. Getting there from a normal shell script requires several deliberate steps — each one correcting a specific way the process could remain accidentally attached to its parent environment.

Problem	Cause	Fix
Daemon killed when terminal closes	Still member of the shell's session; SIGHUP is delivered	Call `setsid()` — first fork + setsid in child
Session leader can acquire a new controlling terminal	First child after setsid() is session leader	Second fork — grandchild can never be session leader
Working directory locks a filesystem (prevents unmount)	CWD inherited from parent	`cd /` in daemon
Inherited file descriptors leak resources or hold files open	All parent FDs carried across fork	Close or redirect stdin/stdout/stderr to `/dev/null`
Inherited umask produces unexpected file permissions	umask from shell	`umask 022` (or your required value)

2 — The Double-Fork Pattern

```bash
#!/usr/bin/env bash
# lib/daemonise.sh — portable double-fork in pure Bash
# Usage: daemonise PIDFILE CMD [ARGS...]

daemonise() {
  local pidfile="$1"; shift

  # ── First fork ──────────────────────────────────────────
  # The parent exits immediately; the shell that launched us
  # considers the job done and returns to the prompt.
  (
    # We are now in a subshell — first child.
    # setsid(2) is not directly callable from Bash, but starting
    # a new process group via a subshell + exec achieves the same
    # effect on Linux when combined with the next fork.
    # Use the 'setsid' utility if available for a true setsid call.

    # ── Second fork ─────────────────────────────────────
    (
      # Grandchild: can never become session leader.
      # Detach from terminal and reset environment.
      cd /
      umask 022

      # Redirect standard file descriptors
      exec 0 /dev/null
      exec 1> /dev/null
      exec 2>&1

      # Write PID file before exec so it's available immediately
      printf '%d\n' "$$" > "$pidfile"

      # Replace grandchild with the actual daemon command
      exec "$@"
    ) &
  ) &
}

# ── Better approach on systems with the 'setsid' utility ────────
daemonise_setsid() {
  local pidfile="$1"; shift
```

```
  # setsid -f forks, calls setsid(2), then execs the command.
  # The double-fork is not strictly needed when setsid -f is used
  # because the child that called setsid() is not the session leader
  # of the new session (the grandchild is). --fork achieves this.
  setsid --fork bash -c "
    cd /
    umask 022
    exec 0/dev/null 2>&1
    printf '%d\n' \"\$\$\" > \"$pidfile\"
    exec \"\$@\"
  " -- "$@"
}
```

Note: Pure-Bash double-forking with two subshells works on Linux but is not a true `setsid(2)` call — the grandchild inherits the grandparent's session unless the grandparent itself called `setsid`. For production use, call `setsid --fork` (util-linux) or write a tiny C wrapper. On systemd systems you should skip all of this and use a `Type=simple` or `Type=forking` service unit instead.

3 — PID Files: Correct Usage

A PID file is a single-line file containing the daemon's PID. It is the canonical way for a start script to later find, signal, and stop the daemon. PID files are simple to get right but easy to get subtly wrong.

```
# ── Write ──────────────────────────────────────────────────
# Write BEFORE exec so the PID is visible as soon as the process exists.
# Use printf, not echo, to avoid platform differences.
printf '%d\n' "$$" > /run/myapp.pid

# ── Read and validate ──────────────────────────────────────
# Never trust a PID file blindly. The process may have died and
# a different process may now hold the same PID.

read_pid() {
  local pidfile="$1"
  [[ -f "$pidfile" ]] || return 1
  local pid
```

```
  read -r pid < "$pidfile"
  # Validate: must be a positive integer
  [[ $pid =~ ^[0-9]+$ ]] || { echo "corrupt pidfile" >&2; return 1; }
  printf '%d' "$pid"
}

pid_is_alive() {
  local pid="$1"
  # kill -0 sends no signal but checks if the process exists and
  # we have permission to signal it.
  kill -0 "$pid" 2>/dev/null
}

pid_is_our_daemon() {
  local pid="$1" name="$2"
  # Cross-check the process name to guard against PID reuse
  local comm
  comm=$(cat "/proc/${pid}/comm" 2>/dev/null)
  [[ "$comm" == "$name" || "$comm" == "${name:0:15}" ]]
  # Linux truncates comm to 15 chars in /proc/PID/comm
}

# ── Cleanup on exit ──────────────────────────────────────
PIDFILE=/run/myapp.pid

cleanup() {
  rm -f "$PIDFILE"
}
trap cleanup EXIT

# ── Locking: prevent two instances ──────────────────────
# The safest way is to hold an exclusive lock on the PID file itself.
# flock -n fails immediately if another process holds the lock.
exec 200>"$PIDFILE"
flock -n 200 || { echo "already running" >&2; exit 1; }
printf '%d\n' "$$" >&200
# The lock is held for the lifetime of the process (FD 200 stays open).
# When the process exits (cleanly or via signal), the OS releases the lock.
# This is more robust than writing then deleting the PID file.
```

4 — Start / Stop / Status Dispatch Pattern

A well-structured init script is a dispatcher: one script, one argument, clean exit codes. The LSB (Linux Standard Base) defines the exit codes that init systems and monitoring tools expect.

Action	Exit 0	Exit 1	Exit 2	Exit 3
`start`	started (or already running)	generic error	invalid argument	—
`stop`	stopped (or already stopped)	generic error	—	—
`status`	running	dead, PID file exists	dead, lock file exists	not running
`restart`	restarted	generic error	—	—

```
#!/usr/bin/env bash
# /etc/init.d/myapp — LSB-compliant init script skeleton
set -euo pipefail

NAME=myapp
DAEMON=/usr/local/bin/myapp
PIDFILE=/run/${NAME}.pid
LOGFILE=/var/log/${NAME}.log
RUNAS=myapp             # run as this user
DAEMON_ARGS="--config /etc/myapp/myapp.conf"

# ── Helper functions ──────────────────────────────────────

get_pid() {
  [[ -f "$PIDFILE" ]] || return 1
  local pid
  read -r pid < "$PIDFILE"
  [[ $pid =~ ^[0-9]+$ ]] && printf '%d' "$pid"
}

is_running() {
  local pid
  pid=$(get_pid) || return 1
  kill -0 "$pid" 2>/dev/null
}

# ── Actions ───────────────────────────────────────────────
```

```
do_start() {
  if is_running; then
    echo "$NAME is already running (pid $(get_pid))"
    return 0
  fi

  echo -n "Starting $NAME... "

  # Drop privileges and daemonise
  start-stop-daemon --start \
    --quiet \
    --background \
    --make-pidfile --pidfile "$PIDFILE" \
    --chuid "$RUNAS" \
    --exec "$DAEMON" \
    -- $DAEMON_ARGS \
    >> "$LOGFILE" 2&1

  # Wait up to 5 s for the PID file to appear
  local i
  for i in $(seq 1 10); do
    is_running && { echo "OK"; return 0; }
    sleep 0.5
  done

  echo "FAILED"
  return 1
}

do_stop() {
  if ! is_running; then
    echo "$NAME is not running"
    rm -f "$PIDFILE"   # clean up stale PID file
    return 0
  fi

  local pid; pid=$(get_pid)
  echo -n "Stopping $NAME (pid $pid)... "

  # Graceful shutdown: SIGTERM, wait, then SIGKILL if needed
```

```
  kill -TERM "$pid"

  local i
  for i in $(seq 1 20); do
    is_running || { rm -f "$PIDFILE"; echo "OK"; return 0; }
    sleep 0.5
  done

  echo -n "(SIGKILL) "
  kill -KILL "$pid" 2>/dev/null
  rm -f "$PIDFILE"
  echo "OK"
}

do_status() {
  if is_running; then
    echo "$NAME is running (pid $(get_pid))"
    return 0     # LSB: 0 = running
  elif [[ -f "$PIDFILE" ]]; then
    echo "$NAME is dead but PID file exists"
    return 1     # LSB: 1 = dead, PID file present
  else
    echo "$NAME is not running"
    return 3     # LSB: 3 = not running, no PID file
  fi
}

do_reload() {
  is_running || { echo "$NAME is not running"; return 1; }
  kill -HUP "$(get_pid)"
  echo "$NAME reloaded"
}

# ── Dispatcher ──────────────────────────────────────────
case "${1:-}" in
  start)   do_start  ;;
  stop)    do_stop   ;;
  restart) do_stop; do_start ;;
  reload)  do_reload ;;
  status)  do_status ;;
  *)
```

```
    echo "Usage: $0 {start|stop|restart|reload|status}" >&2
    exit 2
  ;;
esac
```

5 — Signal Handling in Long-Running Scripts

```
#!/usr/bin/env bash
# A daemon main loop with clean signal handling
set -euo pipefail

PIDFILE=/run/myapp.pid
LOGFILE=/var/log/myapp.log
RUNNING=1
RELOAD=0

log() { printf '[%s] %s\n' "$(date '%F %T')" "$*" >> "$LOGFILE"; }

# ── Signal handlers ──────────────────────────────────────
handle_term() {
  log "SIGTERM received — shutting down"
  RUNNING=0
}

handle_hup() {
  log "SIGHUP received — will reload config on next iteration"
  RELOAD=1
}

handle_usr1() {
  log "SIGUSR1 received — dumping stats"
  dump_stats
}

trap handle_term TERM INT
trap handle_hup  HUP
trap handle_usr1 USR1
```

```
# ── Write PID and start main loop ──────────────────────────
printf '%d\n' "$$" > "$PIDFILE"
trap 'rm -f "$PIDFILE"' EXIT

load_config() { log "loading config"; }
do_work()     { log "working..."; sleep 5; }
dump_stats()  { log "items processed: ${PROCESSED:-0}"; }

load_config
log "started (pid $$)"

while (( RUNNING )); do
  if (( RELOAD )); then
    load_config
    RELOAD=0
  fi
  do_work
done

log "shutdown complete"
```

Signals and their conventional meanings

Signal	Number	Conventional use in daemons
SIGTERM	15	Graceful shutdown — finish current work, clean up, exit
SIGKILL	9	Immediate kill — cannot be caught or ignored
SIGHUP	1	Reload configuration (re-read config file without restart)
SIGUSR1	10	User-defined — dump stats, rotate logs, toggle debug
SIGUSR2	12	User-defined — second custom action
SIGINT	2	Interactive interrupt (Ctrl-C) — often same as SIGTERM

6 — Managing Child Processes

```bash
#!/usr/bin/env bash
# A supervisor: spawn N workers and restart them if they die
set -uo pipefail   # no -e: we handle errors ourselves

WORKERS=4
RUNNING=1
declare -A WORKER_PIDS   # slot → PID

start_worker() {
  local slot="$1"
  # Start the worker in background; record its PID
  /usr/local/bin/worker --slot "$slot" &
  WORKER_PIDS["$slot"]="$!"
  log "slot $slot: started worker pid ${WORKER_PIDS[$slot]}"
}

stop_all() {
  log "stopping all workers"
  local slot
  for slot in "${!WORKER_PIDS[@]}"; do
    kill -TERM "${WORKER_PIDS[$slot]}" 2>/dev/null
  done
  wait   # wait for all children to exit
  log "all workers stopped"
}

trap 'RUNNING=0; stop_all' TERM INT

# ── Spawn initial workers ──────────────────────────────
for slot in $(seq 0 $(( WORKERS - 1 ))); do
  start_worker "$slot"
done

# ── Supervisor loop: restart dead workers ──────────────
while (( RUNNING )); do
  for slot in "${!WORKER_PIDS[@]}"; do
    local pid="${WORKER_PIDS[$slot]}"
    # If the worker exited, wait -n would consume its status.
    # Instead, use kill -0 to check liveness.
    if ! kill -0 "$pid" 2>/dev/null; then
      wait "$pid"    # reap the zombie
```

```
      log "slot $slot: worker $pid exited (rc=$?), restarting"
      start_worker "$slot"
    fi
  done
  sleep 1
done
```

7 — Integrating with systemd

On modern Linux, you should not write a double-fork daemon at all. systemd manages process lifetimes natively and handles PID tracking, logging, dependency ordering, socket activation, and auto-restart. Write a simple foreground process and let systemd do the rest.

Type=simple — the common case

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Application Daemon
After=network.target
Wants=network.target

[Service]
Type=simple
User=myapp
Group=myapp
WorkingDirectory=/var/lib/myapp
ExecStart=/usr/local/bin/myapp --foreground --config /etc/myapp.conf
ExecReload=/bin/kill -HUP $MAINPID
ExecStop=/bin/kill -TERM $MAINPID
PIDFile=/run/myapp.pid      # optional with Type=simple
Restart=on-failure
RestartSec=5s
TimeoutStopSec=30s

# Security hardening
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=strict
```

```
ProtectHome=true
ReadWritePaths=/var/lib/myapp /var/log/myapp

[Install]
WantedBy=multi-user.target
```

Type=notify — sd_notify integration

With `Type=notify`, systemd waits for your process to send a `READY=1` notification before marking it as started. This is the correct type when your daemon needs to initialise (open sockets, load caches) before accepting connections — it prevents dependent services from starting too early.

```
#!/usr/bin/env bash
# Daemon that uses sd_notify to report readiness
# systemd-notify is part of systemd; it writes to the notification socket.

sd_notify() {
  # NOTIFY_SOCKET is set by systemd. If not set, we're not under systemd
  # — silently do nothing so the script works standalone too.
  [[ -S "${NOTIFY_SOCKET:-}" ]] || return 0
  systemd-notify "$@"
}

log() { printf '%s\n' "$*"; }   # systemd captures stdout to the journal

# ── Initialisation phase ──────────────────────────────────
log "Starting up..."
sd_notify "STATUS=Initialising..."

# Simulate slow startup (load DB, open sockets, etc.)
sleep 2
log "Init complete"

# Tell systemd we are ready
sd_notify "READY=1" "STATUS=Running"

# ── Signal handling ───────────────────────────────────────
RUNNING=1
trap 'RUNNING=0' TERM INT
```

```
# ── Main loop ──────────────────────────────────────────────
COUNT=0
while (( RUNNING )); do
  (( COUNT++ )) || true
  sd_notify "STATUS=Processed $COUNT items"
  # Watchdog: tell systemd we are still alive
  sd_notify "WATCHDOG=1"
  sleep 10
done

sd_notify "STOPPING=1" "STATUS=Shutting down"
log "Shutdown complete"
```

```
# Service unit for Type=notify with watchdog
[Service]
Type=notify
NotifyAccess=main         # only main PID may send notifications
WatchdogSec=30s           # kill and restart if no WATCHDOG=1 within 30s
ExecStart=/usr/local/bin/myapp
```

Useful systemctl and journalctl commands

```
# Service lifecycle
systemctl daemon-reload             # re-read unit files after editing
systemctl enable  myapp             # start at boot
systemctl disable myapp             # do not start at boot
systemctl start   myapp
systemctl stop    myapp
systemctl restart myapp
systemctl reload  myapp             # sends ExecReload signal
systemctl status  myapp             # one-line summary + recent log
systemctl is-active  myapp          # exits 0 if running
systemctl is-enabled myapp          # exits 0 if enabled at boot

# Journal (logs)
journalctl -u myapp                 # all logs for the unit
journalctl -u myapp -f              # follow (tail -f equivalent)
journalctl -u myapp --since "1 hour ago"
journalctl -u myapp -n 100          # last 100 lines
```

```
journalctl -u myapp -p err          # errors and above only

# Inspect a unit
systemctl cat    myapp              # show the unit file
systemctl show   myapp              # all properties as key=value
systemctl list-dependencies myapp   # dependency tree
```

8 — Transient Systemd Units with systemd-run

You can launch a one-off or temporary service without writing a unit file using `systemd-run`. This is useful for testing, for wrapping cron jobs with resource limits, and for running scripts under systemd's cgroup management without permanent installation.

```
# Run a command as a transient service (foreground, output to terminal)
systemd-run --pty --same-dir --wait --collect /path/to/script.sh

# Run in background (service unit auto-named)
systemd-run --unit=my-job /path/to/script.sh
systemctl status my-job
journalctl -u my-job -f

# With resource limits — run at idle priority, max 512 MB RAM
systemd-run --nice=19 \
  --property=MemoryMax=512M \
  --property=CPUWeight=10 \
  /path/to/heavy_job.sh

# Timer: run at a specific time (replaces cron)
systemd-run --on-calendar="*-*-* 02:00:00" \
  --unit=nightly-backup \
  /usr/local/bin/backup.sh
```

Exercises

Exercise 1 — Write a robust PID-file library

Implement a sourced Bash library `lib/pidfile.sh` providing these five functions:

- `pidfile_acquire PIDFILE` — write our PID, fail if another process holds a lock
- `pidfile_release PIDFILE` — remove the PID file and release the lock
- `pidfile_read PIDFILE` — print the PID; return 1 if file absent or corrupt
- `pidfile_is_running PIDFILE` — return 0 if the process is alive
- `pidfile_stale PIDFILE` — return 0 if the file exists but the process is dead

Use `flock` for the exclusive lock so the lock is released automatically on abnormal exit. Include a usage example that registers `pidfile_release` on the EXIT trap.

```
#!/usr/bin/env bash
# lib/pidfile.sh — source this file; do not execute it directly

# Internal: FD used for flock (one per pidfile path via name mangling)
_pidfile_fd_for() {
  # Map a path to a stable FD number (200-254 range)
  # For simplicity, use a single global FD; real code would use an assoc
array.
  printf '200'
}

pidfile_acquire() {
  local pidfile="$1"
  local fd; fd=$(_pidfile_fd_for "$pidfile")

  # Open the file on the chosen FD (creates it if absent)
  eval "exec ${fd}>'${pidfile}'"

  # Try an exclusive, non-blocking lock
  if ! flock -n "$fd"; then
    local other_pid
    other_pid=$(pidfile_read "$pidfile" 2>/dev/null)
    printf '%s: already running (pid %s)\n' \
      "$pidfile" "${other_pid:-(unknown)}" >&2
    return 1
  fi

  # Write our PID into the file
  local tmp; tmp="$pidfile.tmp.$$"
```

```
  printf '%d\n' "$$" > "$tmp"
  mv -f "$tmp" "$pidfile"
}

pidfile_release() {
  local pidfile="$1"
  local fd; fd=$(_pidfile_fd_for "$pidfile")
  rm -f "$pidfile"
  eval "exec ${fd}>&-"   # close FD — OS releases the flock
}

pidfile_read() {
  local pidfile="$1"
  [[ -f "$pidfile" ]] || return 1
  local pid
  read -r pid < "$pidfile"
  [[ $pid =~ ^[0-9]+$ ]] || { printf 'corrupt pidfile: %s\n' "$pidfile" >
&2; return 1; }
  printf '%d' "$pid"
}

pidfile_is_running() {
  local pidfile="$1"
  local pid
  pid=$(pidfile_read "$pidfile") || return 1
  kill -0 "$pid" 2>/dev/null
}

pidfile_stale() {
  local pidfile="$1"
  [[ -f "$pidfile" ]] || return 1   # no file → not stale
  pidfile_is_running "$pidfile" && return 1  # still running → not stale
  return 0   # file exists, process dead → stale
}

# ── Usage example ──────────────────────────────────────────
#
#   source lib/pidfile.sh
#   PIDFILE=/run/myapp.pid
#   pidfile_acquire "$PIDFILE" || exit 1
```

```
#   trap 'pidfile_release "$PIDFILE"' EXIT
#   # ... main logic ...
```

Exercise 2 — Complete init script with all LSB actions

Write a complete `/etc/init.d/myworker` init script for a daemon called `myworker` that runs as user `worker`. The script must implement all LSB actions: `start`, `stop`, `restart`, `reload`, `force-reload`, `status`, and `try-restart` (restart only if currently running). Use `start-stop-daemon` for start. Implement correct LSB exit codes for status. The stop action must wait for the process to exit gracefully (SIGTERM, 10 s timeout) before sending SIGKILL.

```
#!/usr/bin/env bash
### BEGIN INIT INFO
# Provides:          myworker
# Required-Start:    $network $local_fs
# Required-Stop:     $network $local_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: My Worker Daemon
### END INIT INFO

NAME=myworker
DAEMON=/usr/local/bin/$NAME
PIDFILE=/run/${NAME}.pid
LOGFILE=/var/log/${NAME}.log
RUNAS=worker
ARGS="--config /etc/myworker.conf"
STOP_TIMEOUT=10

get_pid() {
  [[ -f "$PIDFILE" ]] || return 1
  local p; read -r p < "$PIDFILE"
  [[ $p =~ ^[0-9]+$ ]] && printf '%d' "$p"
}

is_running() {
  local pid; pid=$(get_pid) || return 1
  kill -0 "$pid" 2>/dev/null
```

```
}

do_start() {
  if is_running; then
    echo "$NAME already running (pid $(get_pid))"; return 0
  fi
  echo -n "Starting $NAME... "
  start-stop-daemon --start --quiet --background \
    --make-pidfile --pidfile "$PIDFILE" \
    --chuid "$RUNAS" --exec "$DAEMON" -- $ARGS \
    >> "$LOGFILE" 2&1 || { echo "FAILED"; return 1; }
  local i; for i in $(seq 1 20); do
    is_running && { echo "OK"; return 0; }; sleep 0.5
  done
  echo "FAILED (did not start)"; return 1
}

do_stop() {
  if ! is_running; then
    echo "$NAME is not running"; rm -f "$PIDFILE"; return 0
  fi
  local pid; pid=$(get_pid)
  echo -n "Stopping $NAME (pid $pid)... "
  kill -TERM "$pid"
  local i
  for i in $(seq 1 $(( STOP_TIMEOUT * 2 ))); do
    is_running || { rm -f "$PIDFILE"; echo "OK"; return 0; }
    sleep 0.5
  done
  echo -n "(SIGKILL) "
  kill -KILL "$pid" 2>/dev/null
  rm -f "$PIDFILE"; echo "OK"
}

do_status() {
  if   is_running;      then echo "$NAME running (pid $(get_pid))";
return 0
  elif [[ -f "$PIDFILE" ]]; then echo "$NAME dead but PID file exists"; r
eturn 1
  else                       echo "$NAME not running";
return 3
```

```
  fi
}

case "${1:-}" in
  start)        do_start ;;
  stop)         do_stop ;;
  restart)      do_stop; do_start ;;
  reload|force-reload)
                is_running || { echo "not running"; exit 1; }
                kill -HUP "$(get_pid)"; echo "reloaded" ;;
  status)       do_status ;;
  try-restart)  is_running && { do_stop; do_start; } || true ;;
  *)
    printf 'Usage: %s {start|stop|restart|reload|force-reload|status|try-restart}\n' \
      "$0" >&2; exit 2 ;;
esac
```

Exercise 3 — Foreground daemon with sd_notify and watchdog

Write a script `bin/poller.sh` that:

- Loops every 30 seconds, reading URLs from `/etc/poller/urls.conf` and checking each with `curl -sf`
- Logs results to the journal (stdout, one line per URL)
- Sends `READY=1` after completing its first poll round
- Sends `WATCHDOG=1` after each round (only if `$WATCHDOG_USEC` is set)
- Updates `STATUS=` with the count of passing/failing URLs
- Handles SIGHUP to re-read the URL list without restarting
- Handles SIGTERM/SIGINT for clean shutdown

Also write the accompanying `poller.service` unit file with `Type=notify`, `WatchdogSec=120s`, and reasonable security hardening directives.

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

```
CONFIG="${POLLER_CONFIG:-/etc/poller/urls.conf}"
INTERVAL=30
RUNNING=1
RELOAD=0
declare -a URLS=()

log() { printf '%s\n' "$*"; }   # systemd captures stdout to journal

sd_notify() {
  [[ -S "${NOTIFY_SOCKET:-}" ]] || return 0
  systemd-notify "$@"
}

watchdog_ping() {
  [[ -n "${WATCHDOG_USEC:-}" ]] && sd_notify "WATCHDOG=1"
}

load_config() {
  URLS=()
  if [[ ! -f "$CONFIG" ]]; then
    log "WARN: config not found: $CONFIG"; return
  fi
  while IFS= read -r line; do
    [[ "$line" =~ ^[[:space:]]*(#|$) ]] && continue
    URLS+=("$line")
  done < "$CONFIG"
  log "Loaded ${#URLS[@]} URLs from $CONFIG"
}

poll_urls() {
  local ok=0 fail=0
  local url
  for url in "${URLS[@]}"; do
    if curl -sf --max-time 10 "$url" >/dev/null; then
      log "OK   $url"; (( ok++ )) || true
    else
      log "FAIL $url"; (( fail++ )) || true
    fi
  done
  sd_notify "STATUS=OK:$ok FAIL:$fail"
}
```

```
# ── Signals ──────────────────────────────────────────────
trap 'RUNNING=0'  TERM INT
trap 'RELOAD=1'   HUP

# ── Startup ──────────────────────────────────────────────
load_config
log "Poller starting — $INTERVALs interval"
poll_urls         # first poll before READY=1
sd_notify "READY=1" "STATUS=Running"

# ── Main loop ────────────────────────────────────────────
while (( RUNNING )); do
  if (( RELOAD )); then
    load_config; RELOAD=0
  fi
  sleep "$INTERVAL"
  (( RUNNING )) || break
  poll_urls
  watchdog_ping
done

sd_notify "STOPPING=1"
log "Poller stopped"
```

```
# poller.service
[Unit]
Description=URL Health Poller
After=network-online.target
Wants=network-online.target

[Service]
Type=notify
NotifyAccess=main
User=poller
ExecStart=/usr/local/bin/poller.sh
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure
RestartSec=10s
WatchdogSec=120s
```

```
TimeoutStartSec=60s
TimeoutStopSec=15s

# Hardening
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=strict
ProtectHome=true
ReadOnlyPaths=/etc/poller
CapabilityBoundingSet=      # drop all capabilities

[Install]
WantedBy=multi-user.target
```

Exercise 4 — Supervisor with BATS tests

Write a supervisor script `bin/supervise.sh COMMAND [ARGS...]` that:

- Starts the given command as a child process
- Restarts it automatically if it exits with non-zero status (up to 5 times total, with exponential back-off: 1s, 2s, 4s, 8s, 16s)
- Exits with status 0 if the command exits cleanly (status 0)
- Exits with status 1 after 5 failed restarts
- Propagates SIGTERM to the child and exits cleanly when it receives SIGTERM itself

Then write a BATS test file covering all four cases (clean exit, restart on failure, max-retries exhausted, SIGTERM propagation) using PATH-mocked helper scripts as the supervised command.

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

(( $# >= 1 )) || { printf 'Usage: supervise.sh COMMAND [ARGS...]\n' >&2;
exit 2; }

MAX_RETRIES=5
CHILD_PID=""
```

```
TERMINATING=0

log() { printf '[supervise] %s\n' "$*"; }

handle_term() {
  TERMINATING=1
  log "SIGTERM received — stopping child $CHILD_PID"
  [[ -n "$CHILD_PID" ]] && kill -TERM "$CHILD_PID" 2>/dev/null
}
trap handle_term TERM INT

attempt=0
while (( attempt <= MAX_RETRIES )); do
  (( TERMINATING )) && { log "terminated cleanly"; exit 0; }

  (( attempt > 0 )) && log "restart attempt $attempt/$MAX_RETRIES"

  "$@" &
  CHILD_PID="$!"
  wait "$CHILD_PID"
  rc="$?"
  CHILD_PID=""

  (( TERMINATING )) && { log "terminated cleanly"; exit 0; }
  (( rc == 0    )) && { log "command exited cleanly"; exit 0; }

  log "command exited with rc=$rc"
  (( attempt >= MAX_RETRIES )) && break

  local wait_s; wait_s=$(( 1 << attempt ))  # 1,2,4,8,16
  log "backing off $wait_ss"
  sleep "$wait_s"
  (( attempt++ ))
done

log "max retries ($MAX_RETRIES) exhausted — giving up"
exit 1
```

```
#!/usr/bin/env bats
# test/integration/supervise_test.bats
```

```
setup() {
  load '../test_helper/bats-support/load'
  load '../test_helper/bats-assert/load'
  SCRIPT="${BATS_TEST_DIRNAME}/../../bin/supervise.sh"
  MOCK_DIR="${BATS_TEST_TMPDIR}/mock"
  mkdir -p "$MOCK_DIR"
}

@test "clean exit: supervisor exits 0 immediately" {
  # Command that exits 0
  printf '#!/usr/bin/env bash\nexit 0\n' > "$MOCK_DIR/success_cmd"
  chmod +x "$MOCK_DIR/success_cmd"

  run bash "$SCRIPT" "$MOCK_DIR/success_cmd"
  assert_success
  assert_output --partial "exited cleanly"
}

@test "restart on failure: command fails twice then succeeds" {
  # Fails first 2 attempts, succeeds on 3rd
  printf '#!/usr/bin/env bash
C="%s/c.count"
n=$(cat "$C" 2>/dev/null || echo 0)
echo $(( n+1 )) > "$C"
(( n < 2 )) && exit 1 || exit 0
' "$MOCK_DIR" > "$MOCK_DIR/flaky"
  chmod +x "$MOCK_DIR/flaky"

  # Patch out sleep to speed up the test
  printf '#!/usr/bin/env bash\nexit 0\n' > "$MOCK_DIR/sleep"
  chmod +x "$MOCK_DIR/sleep"
  export PATH="$MOCK_DIR:$PATH"

  run bash "$SCRIPT" "$MOCK_DIR/flaky"
  assert_success
  assert_output --partial "restart attempt"
}

@test "max retries exhausted: exits 1" {
  printf '#!/usr/bin/env bash\nexit 1\n' > "$MOCK_DIR/always_fail"
```

```
  chmod +x "$MOCK_DIR/always_fail"
  printf '#!/usr/bin/env bash\nexit 0\n' > "$MOCK_DIR/sleep"
  chmod +x "$MOCK_DIR/sleep"
  export PATH="$MOCK_DIR:$PATH"

  run bash "$SCRIPT" "$MOCK_DIR/always_fail"
  assert_failure
  assert_output --partial "exhausted"
}

@test "SIGTERM propagates to child and supervisor exits 0" {
  # Long-running child that records whether it got SIGTERM
  printf '#!/usr/bin/env bash
trap "touch %s/got_term; exit 0" TERM
sleep 60 &
wait
' "$MOCK_DIR" > "$MOCK_DIR/long_cmd"
  chmod +x "$MOCK_DIR/long_cmd"

  # Start supervisor in background, send SIGTERM after short delay
  bash "$SCRIPT" "$MOCK_DIR/long_cmd" &
  SUP_PID="$!"
  sleep 0.5
  kill -TERM "$SUP_PID"
  wait "$SUP_PID"
  sup_rc="$?"

  assert_equal "$sup_rc" "0"
  assert_file_exists "$MOCK_DIR/got_term" || true   # best-effort
}
```

Chapter 10 — Dynamic Dispatch and Metaprogramming

Metaprogramming is writing code that treats other code as data — inspecting it, generating it, or modifying it at runtime. Bash supports a surprisingly rich set of these techniques: introspecting function definitions with `declare -f`, calling functions by name stored in variables, building command-dispatch tables, generating functions in loops, and constructing code strings that are safely evaluated. Mastering these patterns produces dramatically shorter, more consistent, and more easily extended scripts — at the cost of code that is harder to follow if the reader is unfamiliar with the idioms. This chapter covers every major technique, explains when each is appropriate, and shows where the sharp edges are.

1 — Introspecting Functions with `declare -f`

`declare -f` prints the definition of a function exactly as Bash has parsed it. `declare -F` prints only the name (and, with `-p`, the source file and line number). These are the building blocks of every introspection-based pattern.

```
# Does this function exist?
function_exists() {
  declare -f "$1" >/dev/null 2&1
}

# Print full definition
greet() { echo "hello $1"; }

declare -f greet
# greet ()
# {
#     echo "hello $1"
# }

# List all defined functions
declare -F
# declare -f greet
# declare -f function_exists
```

```
# ...

# List function names only
declare -F | awk '{print $3}'

# Where was a function defined? (requires extdebug)
shopt -s extdebug
declare -F greet
# greet 1 /path/to/lib.sh    ← name, line, file
shopt -u extdebug

# Copy a function to a new name
function_copy() {
  local src="$1" dst="$2"
  function_exists "$src" || { echo "no such function: $src" >&2; return 1; }
  local body
  # Grab body then replace the name
  body=$(declare -f "$src")
  eval "${dst}() ${body#*()}"
}

greet world          # → hello world
function_copy greet hello_alias
hello_alias world    # → hello world

# Rename a function (copy + unset)
function_rename() {
  function_copy "$1" "$2" && unset -f "$1"
}
```

2 — Calling Functions by Name

The simplest metaprogramming trick in Bash: store a function name in a variable, then call it. Because Bash resolves commands by name at runtime, a variable holding a function name is a first-class function reference.

```
cmd_start()   { echo "starting..."; }
cmd_stop()    { echo "stopping..."; }
cmd_status()  { echo "running"; }
```

```
# Direct call via variable
action=start
"cmd_${action}"        # calls cmd_start

# Validated call — never call an arbitrary variable without validation
dispatch() {
  local action="$1"; shift
  local fn="cmd_${action}"
  if function_exists "$fn"; then
    "$fn" "$@"
  else
    printf 'Unknown action: %s\n' "$action" >&2
    return 1
  fi
}

dispatch start
dispatch stop
dispatch unknown   # → Unknown action: unknown

# Passing a function as a callback (higher-order functions)
apply_to_each() {
  local fn="$1"; shift
  local item
  for item in "$@"; do
    "$fn" "$item"
  done
}

shout() { printf '%s!\n' "${1^^}"; }
apply_to_each shout alpha beta gamma
# ALPHA!
# BETA!
# GAMMA!

# map/filter/reduce in pure Bash using function callbacks
array_map() {
  local fn="$1"; shift
  local -a result=()
  local item
```

```
  for item in "$@"; do
    result+=( "$("$fn" "$item")" )
  done
  printf '%s\n' "${result[@]}"
}

double() { printf '%d' $(( $1 * 2 )); }
array_map double 1 2 3 4
# 2
# 4
# 6
# 8

array_filter() {
  local pred="$1"; shift
  local item
  for item in "$@"; do
    "$pred" "$item" && printf '%s\n' "$item"
  done
}

is_even() { (( $1 % 2 == 0 )); }
array_filter is_even 1 2 3 4 5 6
# 2
# 4
# 6
```

3 — Command-Dispatch Tables

An associative array that maps command names to function names (or to short inline strings) is the Bash equivalent of a jump table. It is cleaner than a long `case` statement and trivially extensible at runtime.

```
# ── Dispatch table via associative array ──────────────────────
declare -A CMDS
CMDS[start]=do_start
CMDS[stop]=do_stop
CMDS[reload]=do_reload
CMDS[status]=do_status
```

```
CMDS[help]=do_help

run_cmd() {
  local name="$1"; shift
  if [[ -v CMDS["$name"] ]]; then
    "${CMDS[$name]}" "$@"
  else
    printf '%s: unknown command "%s"\nAvailable: %s\n' \
      "$0" "$name" "${!CMDS[*]}" >&2
    return 1
  fi
}

# Register a new command at runtime
register_cmd() {
  local name="$1" fn="$2"
  function_exists "$fn" || { echo "no such function: $fn" >&2; return 1; }
  CMDS["$name"]="$fn"
}

# ── Dispatch table with metadata ──────────────────────────
# Store description alongside function name using a compound key
declare -A CMD_FN CMD_DESC

register() {
  local name="$1" fn="$2" desc="$3"
  CMD_FN["$name"]="$fn"
  CMD_DESC["$name"]="$desc"
}

print_help() {
  printf 'Available commands:\n'
  local name
  for name in $(printf '%s\n' "${!CMD_FN[@]}" | sort); do
    printf '  %-15s %s\n' "$name" "${CMD_DESC[$name]}"
  done
}

register start  do_start  "Start the service"
register stop   do_stop   "Stop the service"
```

```
register status do_status "Show service status"
register help   print_help "Show this message"
```

4 — Plugin Systems

A naming convention is all you need for a plugin system in Bash. Define a hook name (e.g. `plugin_init`, `plugin_run`) and source plugin files. Each plugin that defines a function matching the naming convention is automatically discovered and invoked.

```
#!/usr/bin/env bash
# Plugin system: load *.plugin.sh files and call their hooks

PLUGIN_DIR="${PLUGIN_DIR:-/etc/myapp/plugins}"

# ── Load all plugins ──────────────────────────────────────
load_plugins() {
  local file
  for file in "$PLUGIN_DIR"/*.plugin.sh; do
    [[ -f "$file" ]] || continue
    # Snapshot functions before sourcing
    local -A before
    while IFS= read -r line; do
      before["${line##* }"]=1
    done < <(declare -F)

    source "$file"

    # Find newly added functions
    while IFS= read -r line; do
      local fname="${line##* }"
      [[ -v before["$fname"] ]] || \
        printf '[plugin] loaded %s from %s\n' "$fname" "$file"
    done < <(declare -F)
  done
}

# ── Invoke a hook in all plugins that define it ──────────
run_hook() {
  local hook="$1"; shift
```

```
  local fn
  # List all functions matching plugin_*_HOOKNAME pattern
  while IFS= read -r fn; do
    [[ $fn =~ ^plugin_[a-z_]+_${hook}$ ]] || continue
    printf '[hook:%s] %s\n' "$hook" "$fn"
    "$fn" "$@"
  done < <(declare -F | awk '{print $3}')
}

# ── Example plugin file: /etc/myapp/plugins/slack.plugin.sh ──────
#
#   plugin_slack_init()  { echo "Slack plugin loaded"; }
#   plugin_slack_run()   { curl -s "$SLACK_URL" -d "text=$*"; }
#   plugin_slack_finish(){ echo "Slack plugin done"; }
#

load_plugins
run_hook init
run_hook run   "build completed"
run_hook finish
```

5 — Generating Functions at Runtime

When a set of functions differ only in a parameter — a resource name, a prefix, a verb — generate them in a loop instead of copy-pasting. This is the Bash equivalent of a factory function or decorator.

```
# ── Generate CRUD functions for multiple resource types ──────────
for resource in user group project; do
  eval "
    ${resource}_create() {
      printf 'Creating %s: %%s\n' \"${resource}\" \"\$1\"
    }
    ${resource}_delete() {
      printf 'Deleting %s: %%s\n' \"${resource}\" \"\$1\"
    }
    ${resource}_list() {
      printf 'Listing all %ss\n' \"${resource}\"
    }
  "
```

```
done

user_create alice     # → Creating user: alice
group_delete admins   # → Deleting group: admins
project_list          # → Listing all projects

# ── Safer alternative: printf-built function strings ──────────
# Avoid embedding $resource directly in eval strings when the value
# comes from outside the script — use printf %q to quote it.
make_getter() {
  local name="$1" value="$2"
  # printf %q produces a shell-quoted string safe for eval
  local qval; printf -v qval '%q' "$value"
  eval "get_${name}() { printf '%s' ${qval}; }"
}

make_getter hostname "webserver-01"
make_getter region   "us-east-1"

get_hostname    # → webserver-01
get_region      # → us-east-1

# ── Memoisation decorator ─────────────────────────────────────
# Wrap any function so its results are cached by argument
memoize() {
  local fn="$1"
  function_exists "$fn" || { echo "no such function: $fn" >&2; return 1; }

  # Copy original to __orig_FUNCNAME
  function_copy "$fn" "__orig_${fn}"

  # Replace fn with a caching wrapper
  eval "
    declare -A __memo_${fn}
    ${fn}() {
      local _key=\"\$*\"
      local -n _cache=__memo_${fn}
      if [[ -v _cache[\"\$_key\"] ]]; then
        printf '%s' \"\${_cache[\$_key]}\"
        return
      fi
```

```
      local _result
      _result=\$( __orig_${fn} \"\$@\" )
      _cache[\"\$_key\"]=\"\$_result\"
      printf '%s' \"\$_result\"
    }
  "
}

slow_upper() { sleep 1; printf '%s' "${1^^}"; }
memoize slow_upper

slow_upper hello   # takes ~1s
slow_upper hello   # instant — cache hit
slow_upper world   # takes ~1s — different key
```

6 — Safe eval Patterns

`eval` is necessary for some metaprogramming tasks in Bash, but it is the single most dangerous command in the language. Every unsanitised piece of data that reaches `eval` is a command-injection vulnerability. Use these patterns to stay safe.

```
# ── Rule 1: never eval externally-sourced data directly ──────────

# WRONG: $name comes from user input or a file
eval "${name}_init()"   # name='; rm -rf /' → catastrophic

# RIGHT: validate with an allowlist before eval
[[ $name =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]] || { echo "bad name" >&2; exit 1; }
eval "${name}_init() { echo initialised; }"

# ── Rule 2: use printf %q for values that cannot be allowlisted ──
safe_set_var() {
  local varname="$1" value="$2"
  # Validate name is a legal identifier
  [[ $varname =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]] || return 1
  local qval
  printf -v qval '%q' "$value"   # escapes all shell metacharacters
  eval "$varname=$qval"
}
```

```
safe_set_var greeting "hello world; rm -rf /"
echo "$greeting"    # → hello world; rm -rf /   (treated as literal string)

# ── Rule 3: use declare -p round-trip for structured data ──────────
# declare -p produces output that eval can safely consume because
# Bash itself generates it — no untrusted input involved.
declare -A original=([key1]=val1 [key2]="value with spaces")

# Serialise
serialised=$(declare -p original)
printf '%s\n' "$serialised"
# declare -A original=([key1]="val1" [key2]="value with spaces" )

# Deserialise into a new name
eval "${serialised/original/restored}"
declare -p restored
# declare -A restored=([key1]="val1" [key2]="value with spaces" )

# ── Rule 4: prefer declare -n (namerefs) over eval for indirection
# nameref is safe — it only allows variable access, not code execution

# AVOID (eval for indirection):
varname=myarray
eval "echo \${$varname[@]}"

# PREFER (nameref):
declare -n ref=$varname
echo "${ref[@]}"    # no eval, no injection risk
```

7 — Namerefs: Safe Indirect Variable Access

`declare -n` creates a nameref — a variable that is an alias for another variable whose name is its value. Namerefs let you pass array names to functions without using `eval`, and they support all compound-variable operations.

```
# ── Basic nameref ──────────────────────────────────────────
original="hello"
declare -n ref=original
```

```
echo "$ref"         # → hello
ref="world"
echo "$original"    # → world  (same storage)

# ── Passing arrays by name ──────────────────────────────────
array_sum() {
  declare -n _arr="$1"   # _arr is an alias for the named array
  local sum=0
  local x
  for x in "${_arr[@]}"; do
    (( sum += x ))
  done
  printf '%d' "$sum"
}

numbers=(10 20 30 40)
array_sum numbers    # → 100

# ── Returning values via nameref (avoids subshell) ──────────
parse_kv() {
  # $1 = name of associative array to populate
  # $2 = string of key=value pairs (space-separated)
  declare -n _out="$1"
  local pair k v
  for pair in $2; do
    k="${pair%%=*}"
    v="${pair#*=}"
    _out["$k"]="$v"
  done
}

declare -A config
parse_kv config "host=localhost port=5432 dbname=mydb"
printf 'host=%s port=%s\n' "${config[host]}" "${config[port]}"
# host=localhost port=5432

# ── Nameref caution: circular reference ─────────────────────
a=b
declare -n b=a   # b → a → b: circular!
# Bash detects this and prints a warning; $b evaluates to empty.
```

```
# ── Use a unique prefix for internal namerefs in functions ──────
# If a function uses -n ref=$1 and the caller passes "ref" as the
# array name, there is a self-reference collision. Use an unlikely
# prefix (double underscore + function name) to avoid it.
safe_push() {
  declare -n __safe_push_arr="$1"; shift
  __safe_push_arr+=("$@")
}
```

8 — Generating Code at Runtime

Sometimes the right output of a script is another script. Code generation — emitting shell code as text — is a legitimate and powerful technique for configuration management, test fixtures, and build systems.

```
# ── Emit a self-contained script from a template ────────────────
generate_deploy_script() {
  local env="$1" version="$2" outfile="$3"

  # Validate inputs before embedding in generated code
  [[ $env     =~ ^[a-z]+$      ]] || { echo "bad env"     >&2; return 1; }
  [[ $version =~ ^[0-9]+\.[0-9]+\.[0-9]+$ ]] || { echo "bad version" >&2; retu
rn 1; }

  # Heredoc with NO quoting on the delimiter = variable expansion
  # Heredoc with QUOTED delimiter ('EOF') = literal output, no expansion
  cat > "$outfile" <<EOF
#!/usr/bin/env bash
# Auto-generated deploy script
# Generated: $(date -u '+%F %T UTC')
set -euo pipefail
ENV=${env}
VERSION=${version}
echo "Deploying v\${VERSION} to \${ENV}"
# ... rest of deploy logic ...
EOF
  chmod +x "$outfile"
}

generate_deploy_script staging 1.4.2 /tmp/deploy_staging.sh
```

```
# ── Emit a config file from variable state ──────────────────────
emit_env_file() {
  # Print all variables matching PREFIX_* as KEY=value env file
  local prefix="$1"
  local var val qval
  while IFS='=' read -r var val; do
    [[ $var == "${prefix}_"* ]] || continue
    printf -v qval '%q' "$val"
    printf '%s=%s\n' "$var" "$qval"
  done < <(declare -x)
}

export APP_HOST=localhost
export APP_PORT=8080
export APP_DB="my database"
emit_env_file APP
# APP_DB=my\ database
# APP_HOST=localhost
# APP_PORT=8080
```

9 — Tracing and Decorating Functions

```
# ── Automatically wrap every function with a timing decorator ─────
trace_all_functions() {
  local fn
  while IFS= read -r fn; do
    # Skip internal and already-wrapped functions
    [[ $fn == __* || $fn == trace_* ]] && continue
    function_copy "$fn" "__orig_${fn}"
    eval "
      ${fn}() {
        local _t0=\${EPOCHREALTIME}
        __orig_${fn} \"\$@\"
        local _rc=\$?
        local _elapsed
        printf -v _elapsed '%.3f' \"\$(echo \"\${EPOCHREALTIME} - \${_t0}\" |
bc)\"
```

```
        printf '[trace] %s() %.3fs rc=%d\n' '${fn}' \"\${_elapsed}\" \"\${_rc}
\" >&2
        return \$_rc
      }
    "
  done < <(declare -F | awk '{print $3}')
}

# ── before/after hooks on a specific function ──────────────────
add_before_hook() {
  local fn="$1" hook="$2"
  function_copy "$fn" "__orig_${fn}"
  eval "${fn}() { ${hook} \"\$@\"; __orig_${fn} \"\$@\"; }"
}

add_after_hook() {
  local fn="$1" hook="$2"
  function_copy "$fn" "__orig_${fn}"
  eval "
    ${fn}() {
      __orig_${fn} \"\$@\"
      local _rc=\$?
      ${hook} \"\$@\"
      return \$_rc
    }
  "
}

deploy()         { echo "deploying..."; }
pre_deploy_log() { echo "[audit] deploy started by $USER"; }
post_deploy_log(){ echo "[audit] deploy finished"; }

add_before_hook deploy pre_deploy_log
add_after_hook  deploy post_deploy_log

deploy
# [audit] deploy started by alice
# deploying...
# [audit] deploy finished
```

Exercises

Exercise 1 — Build a subcommand CLI dispatcher

Write a script `bin/cli.sh` that acts as a dispatcher for subcommands, similar to how `git` or `kubectl` work. Requirements:

- Subcommands are loaded from `lib/commands/` — each file is named `CMD.sh` and must define `cmd_CMD_run()` and optionally `cmd_CMD_help()`
- Running `cli.sh SUBCMD [ARGS...]` sources the right file then calls `cmd_SUBCMD_run "$@"`
- Running `cli.sh help [SUBCMD]` calls `cmd_SUBCMD_help` if defined, or prints a generic list of available subcommands
- Running `cli.sh` with no arguments prints the command list and exits 1
- Unknown subcommands print a helpful error and exit 2

Also write two example command files: `lib/commands/greet.sh` (prints "Hello, NAME!") and `lib/commands/shout.sh` (prints the argument in uppercase).

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

CMD_DIR="${CMD_DIR:-${BASH_SOURCE[0]%/*}/../lib/commands}"

function_exists() { declare -f "$1" >/dev/null 2&1; }

list_commands() {
  local f
  printf 'Available commands:\n'
  for f in "$CMD_DIR"/*.sh; do
    [[ -f "$f" ]] || continue
    printf '  %s\n' "${f##*/}"
    "${f%.sh}"   # prints just the name without .sh
  done | sed 's|.*/||; s|\.sh||' | sort | xargs -I{} printf '  {}\n'
}

load_command() {
  local name="$1"
  [[ $name =~ ^[a-z][a-z0-9_-]*$ ]] || { printf 'Invalid command name\n'
```

```
>&2; return 1; }
  local file="$CMD_DIR/$name.sh"
  [[ -f "$file" ]] || return 1
  source "$file"
}

subcmd="${1:-}"

if [[ -z "$subcmd" ]]; then
  list_commands
  exit 1
fi

if [[ "$subcmd" == "help" ]]; then
  if [[ -n "${2:-}" ]]; then
    load_command "$2" || { printf 'Unknown command: %s\n' "$2" >&2; exit
2; }
    function_exists "cmd_${2}_help" && "cmd_${2}_help" || printf 'No help
for %s\n' "$2"
  else
    list_commands
  fi
  exit 0
fi

if ! load_command "$subcmd"; then
  printf '%s: unknown command "%s"\n' "${0##*/}" "$subcmd" >&2
  exit 2
fi

fn="cmd_${subcmd}_run"
function_exists "$fn" || {
  printf '%s: %s defines no cmd_%s_run function\n' \
    "${0##*/}" "$subcmd.sh" "$subcmd" >&2
  exit 1
}

shift
"$fn" "$@"
```

```bash
# lib/commands/greet.sh
cmd_greet_run() {
  local name="${1:-World}"
  printf 'Hello, %s!\n' "$name"
}
cmd_greet_help() {
  printf 'Usage: cli.sh greet [NAME]\n  Prints a greeting.\n'
}
```

```bash
# lib/commands/shout.sh
cmd_shout_run() {
  local msg="${*:-NOTHING}"
  printf '%s\n' "${msg^^}"
}
cmd_shout_help() {
  printf 'Usage: cli.sh shout MESSAGE\n  Prints MESSAGE in uppercase.\n'
}
```

Exercise 2 — Generic CRUD factory with namerefs

Write a function `make_resource RESOURCE_NAME ARRAY_NAME` that generates four functions for a given resource type: `RESOURCE_add`, `RESOURCE_remove`, `RESOURCE_list`, and `RESOURCE_contains`. All four functions operate on the array variable named by `ARRAY_NAME` using `declare -n`. Then demonstrate by creating `tag_*` functions operating on a `TAGS` array, and `server_*` functions operating on a `SERVERS` array. Do not use `eval` for the array access — only for function definition.

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

function_exists() { declare -f "$1" >/dev/null 2&1; }

make_resource() {
  local res="$1"     # e.g. "tag"
  local arrname="$2"  # e.g. "TAGS"

  # Validate both names are legal identifiers
```

```
  [[ $res     =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]] || { echo "bad resource nam
e" >&2; return 1; }
  [[ $arrname =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]] || { echo "bad array name"
>&2; return 1; }

  eval "
    ${res}_add() {
      declare -n __${res}_arr=${arrname}
      __${res}_arr+=(\"\$1\")
    }

    ${res}_remove() {
      declare -n __${res}_arr=${arrname}
      local item=\$1 i new=()
      for i in \"\${__${res}_arr[@]}\"; do
        [[ \"\$i\" == \"\$item\" ]] || new+=(\"\$i\")
      done
      __${res}_arr=(\"\${new[@]}\")
    }

    ${res}_list() {
      declare -n __${res}_arr=${arrname}
      printf '%s\n' \"\${__${res}_arr[@]}\"
    }

    ${res}_contains() {
      declare -n __${res}_arr=${arrname}
      local item=\$1 i
      for i in \"\${__${res}_arr[@]}\"; do
        [[ \"\$i\" == \"\$item\" ]] && return 0
      done
      return 1
    }
  "
}

# ── Demo ──────────────────────────────────────────────────────────
declare -a TAGS=()
declare -a SERVERS=()

make_resource tag    TAGS
```

```
make_resource server SERVERS

tag_add production
tag_add europe
tag_add production   # duplicate (allowed — remove for set semantics)
tag_list
# production
# europe
# production

tag_remove production
tag_list
# europe
# production

tag_contains europe  && echo "europe: yes"
tag_contains asia    || echo "asia: no"

server_add web-01
server_add web-02
server_list
# web-01
# web-02
```

Exercise 3 — Memoisation library with TTL

Extend the memoisation pattern from Section 5 into a proper library `lib/memo.sh`. The `memo_wrap FUNCTION [TTL_SECONDS]` function should wrap any function so that:

- Results are cached by argument string
- Cached results expire after TTL seconds (default: no expiry)
- A cache miss calls the original function and stores both the result and the timestamp
- `memo_clear FUNCTION` invalidates the entire cache for that function
- `memo_stats FUNCTION` prints hit count, miss count, and current cache size

Demonstrate with a function that simulates a slow API call (`sleep 1`) and show the cache working with a 5-second TTL.

```
#!/usr/bin/env bash
# lib/memo.sh

function_exists() { declare -f "$1" >/dev/null 2&1; }

function_copy() {
  local body
  body=$(declare -f "$1") || return 1
  eval "${2}() ${body#*()}"
}

memo_wrap() {
  local fn="$1"
  local ttl="${2:-0}"   # 0 = no expiry

  function_exists "$fn" || { echo "no such function: $fn" >&2; return 1;
}
  function_copy "$fn" "__memo_orig_${fn}"

  # Per-function cache and stats arrays
  eval "
    declare -A __memo_cache_${fn}
    declare -A __memo_ts_${fn}
    __memo_hits_${fn}=0
    __memo_misses_${fn}=0
  "

  eval "
    ${fn}() {
      local _key=\"\$*\"
      local -n _cache=__memo_cache_${fn}
      local -n _ts=__memo_ts_${fn}
      local _now
      printf -v _now '%d' \"\${EPOCHSECONDS}\"

      if [[ -v _cache[\"\$_key\"] ]]; then
        local _age=$(( _now - _ts[\$_key] ))
        if (( ${ttl} == 0 || _age < ${ttl} )); then
          (( __memo_hits_${fn}++ )) || true
```

```
          printf '%s' \"\${_cache[\$_key]}\"
          return
        fi
      fi

      (( __memo_misses_${fn}++ )) || true
      local _result
      _result=\$( __memo_orig_${fn} \"\$@\" )
      _cache[\"\$_key\"]=\"\$_result\"
      _ts[\"\$_key\"]=\"\$_now\"
      printf '%s' \"\$_result\"
    }
  "
}

memo_clear() {
  local fn="$1"
  eval "__memo_cache_${fn}=(); __memo_ts_${fn}=()"
}

memo_stats() {
  local fn="$1"
  declare -n _c="__memo_cache_${fn}"
  declare -n _h="__memo_hits_${fn}"
  declare -n _m="__memo_misses_${fn}"
  printf '%s: hits=%d misses=%d size=%d\n' "$fn" "$_h" "$_m" "${#_c[@]}"
}

# ── Demo ──────────────────────────────────────────────────────────
fake_api_call() {
  sleep 1
  printf 'result_for_%s' "$1"
}

memo_wrap fake_api_call 5   # 5-second TTL

time fake_api_call foo   # miss → ~1s
time fake_api_call foo   # hit  → instant
time fake_api_call bar   # miss → ~1s
memo_stats fake_api_call
# fake_api_call: hits=1 misses=2 size=2
```

```
sleep 6                  # let cache expire
time fake_api_call foo   # miss again → ~1s
memo_stats fake_api_call
# fake_api_call: hits=1 misses=3 size=2
```

Exercise 4 — Config-driven function generator

Write a script that reads a configuration file in this format:

```
ALIAS greet  "printf 'Hi, %s\n'"
ALIAS bye    "printf 'Goodbye, %s\n'"
ALIAS shout  "printf '%s\n' \"\${1^^}\""
WRAP  greet  audit_log
WRAP  shout  rate_limit
```

The `ALIAS name body` directive creates a function called `name` using the body string (safely, without allowing injection). The `WRAP fn decorator` directive wraps `fn` so that `decorator` is called before it (the decorator receives the same arguments). Validate that both function names and wrapper names are legal identifiers. Include working implementations of `audit_log` and `rate_limit` so the demo runs.

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

function_exists() { declare -f "$1" >/dev/null 2&1; }

function_copy() {
  local body; body=$(declare -f "$1") || return 1
  eval "${2}() ${body#*()}"
}

is_ident() { [[ $1 =~ ^[a-zA-Z_][a-zA-Z0-9_]*$ ]]; }

# ── Decorators ──────────────────────────────────────────
declare -A _RATE_LAST
RATE_LIMIT_SECS=2
```

```
audit_log() {
  printf '[audit] %s called with: %s\n' "${FUNCNAME[1]}" "$*"
}

rate_limit() {
  local fn="${FUNCNAME[1]}"
  local now="${EPOCHSECONDS}"
  local last="${_RATE_LAST[$fn]:-0}"
  if (( now - last < RATE_LIMIT_SECS )); then
    printf '[rate-limit] %s called too quickly, skipping\n' "$fn" >&2
    return 1
  fi
  _RATE_LAST["$fn"]="$now"
}

# ── Config parser ──────────────────────────────────────────
load_config() {
  local file="$1"
  local directive name rest

  while IFS= read -r line; do
    [[ $line =~ ^[[:space:]]*(#|$) ]] && continue
    read -r directive name rest <<< "$line"

    is_ident "$name" || {
      printf 'Invalid identifier: %s\n' "$name" >&2; continue
    }

    case "$directive" in
      ALIAS)
        # rest is the body string — strip surrounding quotes
        local body="${rest#\"}"; body="${body%\"}"
        # Embed body literally — it's from a trusted config file
        # In a hardened implementation, scan body for dangerous patterns
        eval "${name}() { ${body} \"\$@\"; }"
        printf '[config] defined %s()\n' "$name"
        ;;
      WRAP)
        local dec="$rest"
        is_ident "$dec" || { printf 'Invalid decorator: %s\n' "$dec" >&2;
```

```
continue; }
        function_exists "$name" || {
          printf 'WRAP: function not defined yet: %s\n' "$name" >&2; cont
inue
        }
        function_exists "$dec" || {
          printf 'WRAP: decorator not defined: %s\n' "$dec" >&2; continue
        }
        function_copy "$name" "__wrapped_${name}"
        eval "${name}() { ${dec} \"\$@\" || return; __wrapped_${name}
\"\$@\"; }"
        printf '[config] wrapped %s() with %s()\n' "$name" "$dec"
        ;;
      *) printf 'Unknown directive: %s\n' "$directive" >&2 ;;
    esac
  done < "$file"
}

# ── Demo ──────────────────────────────────────────────────────
load_config functions.conf

greet Alice        # [audit] then Hi, Alice
shout hello        # [rate_limit check] then HELLO
shout hello        # rate-limited on rapid second call
bye Bob            # Goodbye, Bob
```

Chapter 11 — The Bash Completion System

Tab completion transforms a CLI tool from something you have to memorise into something you can explore. Bash's programmable completion system lets you attach arbitrary logic to any command: list subcommands, filter flags by context, query live data, or reflect a config file. This chapter builds a complete mental model of how the system works, then shows you how to write completions that are fast, correct, and genuinely useful.

1 — How Bash Completion Works

When you press Tab, Bash calls the completion function registered for the current command. That function reads context variables, calls `compgen` to generate candidate strings, and writes the results into `COMPREPLY`. Bash then either inserts the single match or displays the list.

Variable	Contents
`COMP_WORDS`	Array of words on the current command line
`COMP_CWORD`	Index into `COMP_WORDS` of the word being completed
`COMP_LINE`	The entire current command line as a string
`COMP_POINT`	Cursor position (byte offset into `COMP_LINE`)
`COMP_KEY`	Key that triggered completion (9 = Tab)
`COMPREPLY`	Array you populate with completion candidates
`cur`	Conventional local: `${COMP_WORDS[COMP_CWORD]}` — the partial word being typed
`prev`	Conventional local: `${COMP_WORDS[COMP_CWORD-1]}` — the word before `cur`

```
# Minimal completion function skeleton
_my_tool_complete() {
  local cur="${COMP_WORDS[COMP_CWORD]}"
  local prev="${COMP_WORDS[COMP_CWORD-1]}"

  # Populate COMPREPLY with matching candidates
```

```
    # compgen -W "list of words" -- "$cur"
    # The -- "$cur" suffix filters the list to words starting with $cur
    COMPREPLY=( $(compgen -W "start stop restart status help" -- "$cur") )
}

# Register the function for the command
complete -F _my_tool_complete my_tool

# Now:  my_tool st  →  start  status  stop
```

> **The bash-completion package** (`bash-completion`) is installed on virtually every modern Linux system. It provides: `/usr/share/bash-completion/bash_completion` — the loader and helper functions; `/usr/share/bash-completion/completions/` — per-command completion files; `~/.local/share/bash-completion/completions/` — your own completions, auto-loaded without any `source` call once bash-completion is initialised.

2 — compgen: The Completion Generator

`compgen` is the workhorse. It generates lists of candidates from many sources and filters them against a prefix.

Flag	Generates
`-W "list"`	Words from a space-separated list
`-f`	Filenames (like default completion)
`-d`	Directories only
`-c`	Command names (anything on PATH)
`-b`	Shell built-in names
`-k`	Shell reserved words
`-A function`	Function names defined in the shell
`-A variable`	Shell variable names
`-A user`	Usernames from `/etc/passwd`
`-A hostname`	Hostnames from `/etc/hosts` and `~/.ssh/known_hosts`

`-A service`	Service names from `/etc/services`
`-G "glob"`	Files matching a glob pattern
`-F function`	Run a shell function; use its output as candidates
`-X "pattern"`	Filter: exclude candidates matching the pattern
`-P "prefix"`	Prepend prefix to each candidate
`-S "suffix"`	Append suffix to each candidate

```
# Generate words matching a prefix
compgen -W "alpha beta gamma delta" -- "ga"
# gamma

# All .conf files in /etc
compgen -G "/etc/*.conf" -- "/etc/sy"
# /etc/sysctl.conf

# Filenames only (no directories)
compgen -f -X  -- ""   # only .log files

# Users whose name starts with "a"
compgen -A user -- "a"

# Add a trailing = to option names (useful for --opt=VALUE style)
compgen -W "--format --output --verbose" -S "=" -- "--fo"
# --format=

# Combine sources with printf
COMPREPLY=( $(
  compgen -W "--help --version" -- "$cur"
  compgen -f                    -- "$cur"
) )
```

3 — The `complete` Built-in

`complete` registers what should happen when Tab is pressed after a specific command. The most important forms:

```
# -F FUNCTION — call a function to generate completions
complete -F _my_tool_complete my_tool

# -W "words" — complete from a fixed word list (no function needed)
complete -W "start stop restart status" service

# -f — complete filenames (restore default file completion)
complete -f cat

# -d — complete directories only
complete -d cd

# -c — complete command names
complete -c which

# -A user — complete usernames
complete -A user su chown

# -o options modify completion behaviour
complete -F _my_complete -o default  my_tool
#  -o default  : fall back to default filename completion if COMPREPLY is empt
y
#  -o nospace  : don't append a space after completing a single match
#  -o filenames: treat results as filenames (apply quoting, add trailing /)
#  -o nosort   : don't sort COMPREPLY before displaying
#  -o bashdefault: fall back to bash-default completion, not just filename

# Show all registered completions
complete -p

# Remove a completion
complete -r my_tool
```

4 — A Realistic Multi-Level Completion

Most CLI tools have subcommands with their own flags. The pattern is: check which word position we are in, dispatch to subcommand-specific logic based on what subcommand has already been typed.

```bash
#!/usr/bin/env bash
# Completion for a hypothetical 'deploy' tool
#
# Usage:
#   deploy [--env ENV] [--dry-run] SUBCOMMAND [ARGS]
#
# Subcommands:
#   app    --name NAME --version VER
#   db     migrate | rollback | status
#   cert   renew --domain DOMAIN

_deploy_complete() {
  local cur="${COMP_WORDS[COMP_CWORD]}"
  local prev="${COMP_WORDS[COMP_CWORD-1]}"
  local words=("${COMP_WORDS[@]}")

  # ── Global flags ──────────────────────────────────────
  local global_flags="--env --dry-run --help --version"

  # ── Handle value completions for flags ────────────────
  case "$prev" in
    --env|-e)
      COMPREPLY=( $(compgen -W "staging production canary" -- "$cur") )
      return ;;
    --name)
      # Complete app names from a live source
      local apps
      apps=$(deploy list-apps 2>/dev/null)
      COMPREPLY=( $(compgen -W "$apps" -- "$cur") )
      return ;;
    --domain)
      # Complete known domains from ~/.deploy/domains
      local domains
      domains=$(cat "${HOME}/.deploy/domains" 2>/dev/null)
      COMPREPLY=( $(compgen -W "$domains" -- "$cur") )
      return ;;
  esac

  # ── Find which subcommand (if any) has been typed ─────
  local subcommand=""
  local i
```

```
  for (( i=1; i < COMP_CWORD; i++ )); do
    case "${words[i]}" in
      app|db|cert) subcommand="${words[i]}"; break ;;
    esac
  done

  # ── Dispatch based on subcommand ──────────────────────
  case "$subcommand" in
    "")
      # No subcommand yet — offer subcommands + global flags
      COMPREPLY=( $(compgen -W "app db cert $global_flags" -- "$cur") ) ;;
    app)
      COMPREPLY=( $(compgen -W "--name --version --strategy" -- "$cur") ) ;;
    db)
      COMPREPLY=( $(compgen -W "migrate rollback status --target" -- "$cur") )
;;
    cert)
      COMPREPLY=( $(compgen -W "renew list revoke --domain --dry-run" -- "$cu
r") ) ;;
  esac
}

complete -F _deploy_complete -o nosort deploy
```

5 — The bash-completion Helper Library

The `bash-completion` package provides helper functions that handle edge cases you would otherwise have to solve yourself — word splitting with quotes, cursor inside a quoted word, and others. Always use them when writing completions that will be distributed.

```
# Load the helpers — this is a no-op if already loaded
if [[ -f /usr/share/bash-completion/bash_completion ]]; then
  source /usr/share/bash-completion/bash_completion
fi

_my_tool_complete() {
  local cur prev words cword
  # _init_completion handles:
  #   - setting cur, prev, words, cword correctly
```

```
    #   - returning early if we should not complete (inside a redirect, etc.)
    #   - dealing with quoted words containing spaces
    _init_completion || return

    # _get_comp_words_by_ref is an alternative that fills named variables
    # _get_comp_words_by_ref -n "=:" cur prev words cword
    # The -n "=:" means "do not split on = or :"

    # filedir — smarter than compgen -f, respects current quoting
    # filedir     → all files and dirs
    # filedir log → only .log files and dirs
    case "$prev" in
      --config|-c)
        filedir '@(conf|cfg|ini)'   # extglob pattern
        return ;;
      --output|-o)
        filedir                     # any file
        return ;;
      --log-dir)
        filedir -d                  # directories only
        return ;;
    esac

    COMPREPLY=( $(compgen -W "--config --output --log-dir --verbose --help" \
      -- "$cur") )
  }

  complete -F _my_tool_complete -o default my_tool
```

Key helper functions from bash-completion

Helper	What it does
`_init_completion`	Sets `cur`, `prev`, `words`, `cword`; handles quoting and redirects
`filedir [ext]`	Complete files/dirs, optionally filtered by extension; respects current quoting
`_get_comp_words_by_ref`	Like `_init_completion` but gives control over split characters
`_count_args`	Count positional arguments already on the line (excludes flags)

`_have CMD`	Return 0 if `CMD` is available on PATH
`_longopt CMD`	Auto-generate completions from `CMD --help` output (many tools)

6 — Dynamic Completions from Live Data

The most useful completions query real state. The key discipline is *speed*: if a completion takes more than ~200 ms the user notices. Use caching for data that changes slowly.

```bash
#!/usr/bin/env bash
# Completing from live data with a TTL cache

# ── Cache helper ──────────────────────────────────────────
_cached_compgen() {
  # Usage: _cached_compgen CACHE_KEY TTL_SECS COMMAND [ARGS...]
  # Runs COMMAND, caches output in /tmp, regenerates after TTL_SECS
  local key="$1" ttl="$2"; shift 2
  local cache="/tmp/.completion_cache_${key}"

  if [[ -f "$cache" ]]; then
    local age
    age=$(( EPOCHSECONDS - $(stat -c '%Y' "$cache") ))
    (( age < ttl )) && { cat "$cache"; return; }
  fi

  # Run the command; write to cache in background so Tab feels instant
  "$@" 2>/dev/null | tee "$cache"
}

# ── Docker container completion ──────────────────────────
_docker_containers() {
  _cached_compgen docker_containers 30 \
    docker ps --format '{{.Names}}'
}

_docker_images() {
  _cached_compgen docker_images 60 \
    docker images --format '{{.Repository}}:{{.Tag}}'
}
```

```
_myapp_complete() {
  local cur prev words cword
  _init_completion || return

  case "$prev" in
    --container)
      COMPREPLY=( $(compgen -W "$(_docker_containers)" -- "$cur") )
      return ;;
    --image)
      COMPREPLY=( $(compgen -W "$(_docker_images)" -- "$cur") )
      return ;;
  esac

  COMPREPLY=( $(compgen -W "--container --image --help" -- "$cur") )
}

complete -F _myapp_complete myapp

# ── Git branch completion (no caching needed — git is fast) ──────
_git_branches() {
  git branch --format '%(refname:short)' 2>/dev/null
}

_git_remote_branches() {
  git branch -r --format '%(refname:short)' 2>/dev/null | \
    sed 's|origin/||'
}

_my_deploy_complete() {
  local cur prev words cword
  _init_completion || return

  case "$prev" in
    --branch|-b)
      COMPREPLY=( $(compgen -W "$(_git_branches)" -- "$cur") )
      return ;;
  esac

  COMPREPLY=( $(compgen -W "--branch --env --dry-run --help" -- "$cur") )
}
```

```
complete -F _my_deploy_complete my_deploy
```

7 — Installing and Distributing Completions

```
# ── Drop-in directory (auto-loaded, no source needed) ──────────
# System-wide: completions loaded for all users
/usr/share/bash-completion/completions/my_tool

# Per-user: takes precedence over system completions
${HOME}/.local/share/bash-completion/completions/my_tool

# Both directories are scanned when bash-completion initialises.
# The file must be named exactly after the command, no extension.

# ── Makefile install target ────────────────────────────────────
# PREFIX ?= /usr/local
# install-completion:
#     install -d $(PREFIX)/share/bash-completion/completions
#     install -m644 completions/my_tool \
#         $(PREFIX)/share/bash-completion/completions/my_tool

# ── Lazy loading (completion-on-demand) ────────────────────────
# For large completion files, register a stub that sources the real
# file on first Tab press. After sourcing, re-runs completion.

_my_tool_loader() {
  # Remove the stub
  complete -r my_tool
  # Source the real completion file
  source /usr/local/share/my_tool/completions.bash
  # Re-trigger completion on the current word
  if [[ -n "${BASH_COMPLETION_VERSINFO:-}" ]]; then
    local cur; cur="${COMP_WORDS[COMP_CWORD]}"
    COMPREPLY=( $(compgen -W "" -- "$cur") )
  fi
  # Call the real completion function that was just loaded
  _my_tool_complete
```

```
}

# Register the loader stub instead of the real function
complete -F _my_tool_loader my_tool

# ── Sourcing in .bashrc (fallback for systems without bash-completion)
# if ! shopt -oq posix; then
#   [ -f /etc/bash_completion ] && source /etc/bash_completion
#   [ -f ~/.bash_completion   ] && source ~/.bash_completion
# fi
```

8 — Completion for Subcommand-Style Tools

Tools structured like `git`, `kubectl`, or `cargo` — with many subcommands each having their own flags — benefit from a table-driven approach that scales without repetition.

```
#!/usr/bin/env bash
# Table-driven completion for a tool with many subcommands
# Each subcommand's flags are stored in an associative array.

declare -A _CLI_FLAGS
_CLI_FLAGS[""]="--config --verbose --quiet --help --version"
_CLI_FLAGS[build]="--target --release --debug --features --no-default-feature
s"
_CLI_FLAGS[test]="--filter --timeout --parallel --coverage"
_CLI_FLAGS[deploy]="--env --branch --strategy --dry-run --force"
_CLI_FLAGS[config]="get set unset list"
_CLI_FLAGS[logs]="--tail --follow --since --level --service"

_cli_get_subcommand() {
  # Scan COMP_WORDS[1..COMP_CWORD-1] for a known subcommand
  local i
  for (( i=1; i < COMP_CWORD; i++ )); do
    [[ -v _CLI_FLAGS["${COMP_WORDS[i]}"] ]] && {
      printf '%s' "${COMP_WORDS[i]}"
      return
    }
  done
}
```

```
_cli_complete() {
  local cur prev words cword
  _init_completion || return

  local subcmd
  subcmd=$(_cli_get_subcommand)

  if [[ -z "$subcmd" ]]; then
    # No subcommand yet — offer global flags + all subcommands
    local subcmds
    subcmds="${!_CLI_FLAGS[*]}"
    subcmds="${subcmds//[[:space:]]$'\t'}"  # remove empty key
    COMPREPLY=( $(compgen \
      -W "${_CLI_FLAGS[""]} ${!_CLI_FLAGS[*]}" \
      -- "$cur") )
  else
    # Subcommand is known — offer its flags
    COMPREPLY=( $(compgen \
      -W "${_CLI_FLAGS[$subcmd]:-} ${_CLI_FLAGS[""]}" \
      -- "$cur") )
  fi
}

complete -F _cli_complete -o nosort cli
```

9 — Debugging Completions

```
# ── See what complete is registered for a command ──────────────
complete -p git
# complete -o bashdefault -o default -o nospace -F __git_wrap__git_main git

# ── Manually call a completion function ────────────────────────
# Simulate pressing Tab after "deploy --e"
COMP_WORDS=(deploy --e)
COMP_CWORD=1
COMP_LINE="deploy --e"
COMP_POINT="${#COMP_LINE}"
```

```
_deploy_complete
printf '%s\n' "${COMPREPLY[@]}"

# ── Trace execution ──────────────────────────────────────────
# Set xtrace in just the completion function to see what it does
_deploy_complete() {
  set -x
  # ... function body ...
  set +x
} 2>/tmp/completion_debug.log

# ── FIGNORE: exclude file suffixes from completion ──────────
FIGNORE=".o:.pyc:.class"   # these extensions never appear in file completion

# ── COMP_WORDBREAKS: characters that split words ────────────
# Default: " \t\n\"'><=;|&(:"
# If your tool uses : in identifiers (e.g. kubectl resource:name),
# remove : from COMP_WORDBREAKS:
COMP_WORDBREAKS="${COMP_WORDBREAKS//:/}"
```

Exercises

Exercise 1 — Complete a backup tool

Write a completion function for the following tool:

```
backup.sh [OPTIONS] SOURCE DEST
  Options:
    --compress (-z)  : gzip | bzip2 | xz | none
    --exclude (-e)   : glob pattern (can be given multiple times)
    --dry-run        : no flag value
    --log-file (-l)  : path to a log file
    --retain (-r)    : number of days (integer)
    --help
```

Completion rules: `--compress` should offer the four named values; `--log-file` should complete filenames with `filedir`; `--exclude` should offer common glob patterns (`*.tmp *.log .git node_modules`); `SOURCE` and `DEST` are positional — both should complete directories; all flag

completions should be context-sensitive (no duplication of already-present flags). Use `_init_completion`.

```
#!/usr/bin/env bash
# completions/backup.sh

_backup_complete() {
  local cur prev words cword
  _init_completion || return

  # ── Flag value completions ──────────────────────────────
  case "$prev" in
    --compress|-z)
      COMPREPLY=( $(compgen -W "gzip bzip2 xz none" -- "$cur") )
      return ;;
    --log-file|-l)
      filedir
      return ;;
    --exclude|-e)
      COMPREPLY=( $(compgen -W "'*.tmp' '*.log' '.git' 'node_modules' '*.
pyc' '__pycache__'" \
        -- "$cur") )
      return ;;
    --retain|-r)
      COMPREPLY=( $(compgen -W "7 14 30 60 90" -- "$cur") )
      return ;;
  esac

  # ── Count positional args already on the line ──────────
  local positionals=0
  local i
  for (( i=1; i < cword; i++ )); do
    [[ "${words[i]}" == -* ]] && continue
    # Skip values of flags that take an argument
    [[ "${words[i-1]}" =~ ^(--compress|-z|--log-file|-l|--exclude|-e|--re
tain|-r)$ ]] \
      && continue
    (( positionals++ ))
  done
```

```
  # ── Positional args: SOURCE and DEST are directories ──────────
  if (( positionals < 2 )) && [[ "$cur" != -* ]]; then
    filedir -d
    return
  fi

  # ── Collect already-used non-repeatable flags ─────────────────
  local used=" ${words[*]} "
  local all_flags="--compress -z --exclude -e --dry-run --log-file -l --r
etain -r --help"
  local -a available=()
  local flag
  for flag in $all_flags; do
    # Allow --exclude to be repeated; suppress others once used
    [[ $flag == "--exclude" || $flag == "-e" ]] || \
      [[ "$used" != *" ${flag} "* ]] && available+=("$flag")
  done

  COMPREPLY=( $(compgen -W "${available[*]}" -- "$cur") )
}

complete -F _backup_complete -o default backup.sh backup
```

Exercise 2 — Dynamic completion from a config file

Your tool `svc` manages services defined in `~/.config/svc/services.conf`, one service name per line, with optional `#` comments. Write a completion function that:

- Supports subcommands: `start stop restart status logs tail`
- After a subcommand that takes a service name, reads `~/.config/svc/services.conf` and completes service names (the config file changes rarely — cache it for 60 s)
- `svc logs` and `svc tail` additionally accept `--follow`, `--lines N`, `--since TIMESTAMP`
- `--lines` completes the values `50 100 200 500 all`
- Global flag `--config PATH` completes with `filedir`

```
#!/usr/bin/env bash
# completions/svc
```

```
_svc_services() {
  local conf="${HOME}/.config/svc/services.conf"
  local cache="/tmp/.svc_services_cache"

  if [[ -f "$cache" ]]; then
    local age
    age=$(( EPOCHSECONDS - $(stat -c '%Y' "$cache" 2>/dev/null || echo 0)
))
    (( age < 60 )) && { cat "$cache"; return; }
  fi

  [[ -f "$conf" ]] || return
  grep -v '^[[:space:]]*#' "$conf" | grep -v '^[[:space:]]*$' | tee "$cac
he"
}

_svc_complete() {
  local cur prev words cword
  _init_completion || return

  local subcmds="start stop restart status logs tail"
  local log_flags="--follow --lines --since"

  # Global flag values
  case "$prev" in
    --config)
      filedir; return ;;
    --lines)
      COMPREPLY=( $(compgen -W "50 100 200 500 all" -- "$cur") )
      return ;;
    --since)
      COMPREPLY=( $(compgen -W "1h 6h 24h yesterday today" -- "$cur") )
      return ;;
  esac

  # Find subcommand
  local subcmd=""
  local i
  for (( i=1; i < cword; i++ )); do
    [[ " ${subcmds} " == *" ${words[i]} "* ]] && {
      subcmd="${words[i]}"; break
```

```
    }
  done

  if [[ -z "$subcmd" ]]; then
    COMPREPLY=( $(compgen -W "$subcmds --config --help" -- "$cur") )
    return
  fi

  # Service name position (word after subcommand)
  local service_typed=0
  for (( i=1; i < cword; i++ )); do
    [[ "${words[i]}" == "$subcmd" ]] && {
      [[ "${words[i+1]:-}" != -* && -n "${words[i+1]:-}" ]] \
        && service_typed=1
      break
    }
  done

  if (( ! service_typed )) && [[ "$cur" != -* ]]; then
    local svcs; svcs=$(_svc_services)
    COMPREPLY=( $(compgen -W "$svcs" -- "$cur") )
    return
  fi

  case "$subcmd" in
    logs|tail)
      COMPREPLY=( $(compgen -W "$log_flags" -- "$cur") ) ;;
    *)
      COMPREPLY=() ;;
  esac
}

complete -F _svc_complete -o nosort svc
```

Exercise 3 — Completion generator for git-style tools

Write a reusable function `_make_subcmd_completion TOOL_NAME SUBCOMMAND_FN FLAG_ARRAY_NAME` that installs a complete function for any tool with subcommands, given:

- `TOOL_NAME` — the command name to register

- `SUBCOMMAND_FN` — name of a function that prints available subcommands (one per line); called once and cached for the session
- `FLAG_ARRAY_NAME` — name of an associative array mapping subcommand names to their flag strings (empty key = global flags)

The generated completion function must: offer global flags when no subcommand is typed, offer subcommand-specific flags after a known subcommand, and fall back to `-o default` for unknown subcommands. Demonstrate by using it to add completions for a fictional `proj` tool with subcommands `build test lint run`.

```
#!/usr/bin/env bash
# lib/completion_factory.sh

_make_subcmd_completion() {
  local tool="$1"
  local subcmd_fn="$2"
  local flags_arr="$3"

  # Validate
  [[ $tool      =~ ^[a-zA-Z_][a-zA-Z0-9_-]*$ ]] || { echo "bad tool name"
>&2; return 1; }
  [[ $subcmd_fn =~ ^[a-zA-Z_][a-zA-Z0-9_]*$   ]] || { echo "bad function
name" >&2; return 1; }
  [[ $flags_arr =~ ^[a-zA-Z_][a-zA-Z0-9_]*$   ]] || { echo "bad array nam
e"   >&2; return 1; }

  # Generate a session-scoped subcommand cache variable name
  local cache_var="__compcache_${tool//-/_}"

  # Emit the completion function
  eval "
    _${tool//-/_}_complete() {
      local cur prev words cword
      _init_completion || return

      # Populate subcommand cache on first call
      if [[ -z \"\${${cache_var}:-}\" ]]; then
        ${cache_var}=\$( ${subcmd_fn} )
      fi
      local _subcmds=\"\${${cache_var}}\"
```

```
      # Find typed subcommand
      local _subcmd=\"\" i
      for (( i=1; i < cword; i++ )); do
        if printf '%s\n' \$_subcmds | grep -qxF \"\${words[i]}\"; then
          _subcmd=\"\${words[i]}\"; break
        fi
      done

      # Reference the flags associative array
      declare -n _flags=${flags_arr}

      if [[ -z \"\$_subcmd\" ]]; then
        COMPREPLY=( \$(compgen -W \"\${_flags[\"\"]:+\${_flags[\"\"]}}
\${_subcmds}\" -- \"\$cur\") )
      else
        local _sf=\"\${_flags[\"\$_subcmd\"]:-}\"
        local _gf=\"\${_flags[\"\"]:+\${_flags[\"\"]}}\"
        COMPREPLY=( \$(compgen -W \"\${_sf} \${_gf}\" -- \"\$cur\") )
      fi
    }
    complete -F _${tool//-/_}_complete -o default ${tool}
  "
}

# ── Demo: proj tool ──────────────────────────────────────────
declare -A PROJ_FLAGS
PROJ_FLAGS[""]="--config --verbose --quiet --help"
PROJ_FLAGS[build]="--target --release --debug --watch"
PROJ_FLAGS[test]="--filter --coverage --watch --bail"
PROJ_FLAGS[lint]="--fix --strict --format"
PROJ_FLAGS[run]="--port --host --reload --env"

_proj_subcommands() {
  printf '%s\n' build test lint run
}

_make_subcmd_completion proj _proj_subcommands PROJ_FLAGS
```

Exercise 4 — Complete a custom SSH wrapper

Write completion for a script `sshjump HOST [COMMAND]` that SSH's through a jump host to a target. The host list comes from two sources: `~/.ssh/config` (extract `Host` lines, excluding patterns with `*` or `?`) and `~/.config/sshjump/hosts` (plain list, one per line, with comments). Merge and deduplicate both sources, cache for 120 seconds. After the host argument is satisfied, `COMMAND` should complete with remote command names — offer a fixed set of useful ones (`bash htop journalctl systemctl top`) plus whatever is on the local PATH filtered to single-word commands (use `compgen -c`).

```
#!/usr/bin/env bash
# completions/sshjump

_sshjump_hosts() {
  local cache="/tmp/.sshjump_hosts_cache"

  if [[ -f "$cache" ]]; then
    local age
    age=$(( EPOCHSECONDS - $(stat -c '%Y' "$cache" 2>/dev/null || echo 0)
))
    (( age < 120 )) && { cat "$cache"; return; }
  fi

  {
    # From ~/.ssh/config: Host lines without wildcards
    [[ -f "${HOME}/.ssh/config" ]] && \
      awk '/^[Hh]ost[[:space:]]/{
        for(i=2;i<=NF;i++) if($i !~ /[*?]/) print $i
      }' "${HOME}/.ssh/config"

    # From ~/.config/sshjump/hosts
    [[ -f "${HOME}/.config/sshjump/hosts" ]] && \
      grep -v '^[[:space:]]*\(#\|$\)' "${HOME}/.config/sshjump/hosts"
  } | sort -u | tee "$cache"
}

_sshjump_complete() {
  local cur prev words cword
  _init_completion || return
```

```
  # Count non-flag positional args already typed
  local positionals=0
  local i
  for (( i=1; i < cword; i++ )); do
    [[ "${words[i]}" == -* ]] || (( positionals++ ))
  done

  case "$positionals" in
    0)  # Completing HOST
      local hosts; hosts=$(_sshjump_hosts)
      COMPREPLY=( $(compgen -W "$hosts" -- "$cur") ) ;;
    1)  # Completing COMMAND
      local well_known="bash htop journalctl systemctl top ps df du tail
grep"
      COMPREPLY=( $(compgen -W "$well_known" -- "$cur"
                    compgen -c                  -- "$cur") )
      # Deduplicate
      local -A seen
      local -a uniq=()
      local item
      for item in "${COMPREPLY[@]}"; do
        [[ -v seen["$item"] ]] || { uniq+=("$item"); seen["$item"]=1; }
      done
      COMPREPLY=("${uniq[@]}") ;;
    *)  COMPREPLY=() ;;
  esac
}

complete -F _sshjump_complete -o default sshjump
```

Chapter 12 — Integrating Bash with Other Languages

Bash's real power in a polyglot world is not what it can do alone — it is how cleanly it can orchestrate everything else. Knowing when to hand off to Python, Node, Ruby, or awk, how to pass structured data across that boundary, and how to use another language as a persistent coprocess rather than a repeatedly-forked subprocess are the skills that separate throw-it-together shell scripts from production-grade automation. This final chapter covers every major integration pattern and ends with an honest decision framework for when to stop using Bash at all.

1 — The Fork-per-Call Tax

Every `$(python3 ...)` inside a loop pays a full fork+exec cost — typically 5–30 ms on a modern Linux system. For one-off calls this is irrelevant. For ten thousand iterations it is a 30-second wall clock bill you did not budget for.

```
# Expensive: one Python process per line
while IFS= read -r line; do
  result=$(python3 -c "import sys; print(sys.stdin.read().upper())" <<< "$lin
e")
  printf '%s\n' "$result"
done < bigfile.txt

# Better: one process, all lines
python3 -c "import sys; sys.stdout.write(sys.stdin.read().upper())" \
  < bigfile.txt

# Best for bidirectional work: a persistent coprocess (see Section 2)
```

Pattern	Forks	Use when
One-shot call `$(python3 script.py)`	1	Called once or rarely; simplest code
Batch stdin→stdout pipe	1	Large input, stateless transformation

Coprocess (persistent)	1	Many calls in a loop; stateful interpreter session
Background server + socket	1 + overhead	Multiple shell scripts sharing one interpreter

2 — Persistent Coprocesses

The coprocess pattern (introduced in Chapter 4) becomes especially valuable when the "other language" needs to be initialised once — loading libraries, connecting to a database, compiling a regex — and then serve many requests.

Python coprocess

```
#!/usr/bin/env bash
# A persistent Python worker that handles JSON requests over stdin/stdout

# ── The Python worker script ──────────────────────────────
cat > /tmp/worker.py <<'PYEOF'
import sys, json, hashlib, re

# One-time initialisation: compile regex, load lookup tables, etc.
PATTERN = re.compile(r'\b\w{4,}\b')

for raw in sys.stdin:
    raw = raw.strip()
    if not raw:
        continue
    try:
        req = json.loads(raw)
        op  = req.get('op')

        if op == 'hash':
            result = hashlib.sha256(req['data'].encode()).hexdigest()
        elif op == 'words':
            result = PATTERN.findall(req['text'])
        elif op == 'upper':
            result = req['text'].upper()
        else:
```

```
            result = None

        print(json.dumps({'ok': True,  'result': result}), flush=True)
    except Exception as e:
        print(json.dumps({'ok': False, 'error':  str(e)}),  flush=True)
PYEOF

# ── Start the coprocess ──────────────────────────────────────
coproc PY (python3 -u /tmp/worker.py)
# -u: unbuffered — critical, otherwise our writes are never read

py_call() {
  # Send a JSON request; read the JSON response
  local request="$1"
  printf '%s\n' "$request" >&"${PY[1]}"
  read -r response <&"${PY[0]}"
  printf '%s' "$response"
}

# ── Use it in a loop — only one Python process throughout ─────
while IFS= read -r line; do
  req=$(printf '{"op":"hash","data":"%s"}' \
    "$(printf '%s' "$line" | sed 's/"/\\"/g')")
  resp=$(py_call "$req")
  printf '%s\t%s\n' "$line" "$resp"
done < input.txt

# Shut down the worker
kill "$PY_PID" 2>/dev/null; wait "$PY_PID" 2>/dev/null
```

Node.js coprocess

```
cat > /tmp/worker.js <<'JSEOF'
const readline = require('readline');
const crypto   = require('crypto');
const rl = readline.createInterface({ input: process.stdin });

rl.on('line', (raw) => {
  try {
    const req = JSON.parse(raw);
```

```
    let result;
    if      (req.op === 'md5')    result = crypto.createHash('md5').update(re
q.data).digest('hex');
    else if (req.op === 'upper')  result = req.text.toUpperCase();
    else if (req.op === 'slugify')result = req.text.toLowerCase().replace(/\s
+/g, '-');
    process.stdout.write(JSON.stringify({ ok: true, result }) + '\n');
  } catch(e) {
    process.stdout.write(JSON.stringify({ ok: false, error: e.message }) +
'\n');
  }
});
JSEOF

coproc JS (node /tmp/worker.js)

js_call() {
  printf '%s\n' "$1" >&"${JS[1]}"
  read -r REPLY <&"${JS[0]}"
  printf '%s' "$REPLY"
}

js_call '{"op":"slugify","text":"Hello World"}'
# {"ok":true,"result":"hello-world"}
```

3 — Passing Data Structures via JSON

JSON is the universal interchange format between Bash and other languages. `jq` is the standard tool for reading and writing it from shell scripts.

```
# ── Reading JSON in Bash ──────────────────────────────────────
json='{"name":"alice","roles":["admin","user"],"meta":{"age":30}}'

# Extract a scalar
name=$(jq -r '.name'         <<< "$json")     # -r = raw (no quotes)
age=$(jq  -r '.meta.age'     <<< "$json")
role0=$(jq -r '.roles[0]'    <<< "$json")

# Read a JSON array into a Bash array
```

```
mapfile -t roles < <(
  jq -r '.roles[]' <<< "$json"
)
printf 'role: %s\n' "${roles[@]}"
# role: admin
# role: user

# Iterate over JSON object keys
while IFS='=' read -r key val; do
  printf '%s → %s\n' "$key" "$val"
done < <(jq -r '.meta | to_entries[] | "\(.key)=\(.value)"' <<< "$json")
# age → 30

# ── Building JSON from Bash variables ──────────────────────
# WRONG: string interpolation breaks on special characters
printf '{"name":"%s"}' "$name"    # breaks if name contains " or \

# RIGHT: let jq do the escaping via --arg
payload=$(jq -n \
  --arg   name  "$name"   \
  --argjson age "$age"    \
  '{"name":$name,"age":$age}')

# Build a JSON array from a Bash array
tags=(production europe tier-1)
json_tags=$(printf '%s\n' "${tags[@]}" | jq -R '.' | jq -s '.')
# → ["production","europe","tier-1"]

# Build a JSON object from an associative array
declare -A env_vars=([HOST]=localhost [PORT]=8080 [DB]=mydb)
json_obj=$(for k in "${!env_vars[@]}"; do
  jq -n --arg k "$k" --arg v "${env_vars[$k]}" '{"key":$k,"value":$v}'
done | jq -s 'from_entries')

# ── Streaming JSON (NDJSON / JSON Lines) ──────────────────
# Each line is a self-contained JSON object — safe to pipe, grep, tail
jq -c '.[]' big_array.json | while IFS= read -r obj; do
  name=$(jq -r '.name' <<< "$obj")
  printf 'Processing %s\n' "$name"
done
```

4 — Embedding Other Languages Inline

For short, self-contained logic that does not warrant a separate file, embed the other language directly in the shell script using a heredoc. The heredoc keeps related code together and avoids the file-management overhead of temporary scripts.

```
# ── Inline Python ──────────────────────────────────────────
result=$(python3 <<'EOF'
import json, sys

data = [{"id": i, "val": i**2} for i in range(1, 6)]
print(json.dumps(data))
EOF
)
echo "$result"
# [{"id": 1, "val": 1}, {"id": 2, "val": 4}, ...]

# ── Passing Bash variables into inline Python ──────────────
threshold=42
label="my label"

# Method 1: environment variables (safe for any value)
export THRESHOLD="$threshold"
export LABEL="$label"
python3 <<'EOF'
import os
print(f"threshold={os.environ['THRESHOLD']}, label={os.environ['LABEL']}")
EOF

# Method 2: unquoted heredoc — Bash expands $vars before Python sees them
# ONLY safe when the values are validated (no quotes, no backslashes)
[[ $threshold =~ ^[0-9]+$ ]] || { echo "bad threshold" >&2; exit 1; }
python3 <<EOF
print("threshold is ${threshold}")
EOF

# ── Inline Ruby ────────────────────────────────────────────
csv_data="alice,30,engineer
bob,25,designer"
```

```
ruby <<'EOF'
require 'csv'
STDIN.each_line do |line|
  row = CSV.parse_line(line.chomp)
  puts "#{row[0].upcase} (#{row[2]}), age #{row[1]}"
end
EOF
<<< "$csv_data"

# ── Inline Perl ──────────────────────────────────────────────
# Perl shines for complex text transformations
perl -00 -ne 'print if /ERROR/i' /var/log/app.log
# -00: paragraph mode (blank line = record separator)
# -n: loop over input; -e: inline code

# ── Inline awk — already in most scripts, mentioned for completeness
awk 'BEGIN{OFS="\t"} NR>1{sum+=$3} END{printf "total: %.2f\n", sum}' data.csv
```

5 — Shell as Glue: Orchestrating Multi-Language Pipelines

Bash's most natural role in a polyglot system is as the top-level orchestrator: each stage of a pipeline is handled by the language best suited to it, and Bash wires them together.

```
#!/usr/bin/env bash
# A real-world ETL pipeline:
# 1. Fetch raw data (curl)
# 2. Validate and transform (Python)
# 3. Aggregate stats (awk)
# 4. Format report (jq)
# 5. Upload result (curl)
set -euo pipefail

API_URL="https://api.example.com/events"
TOKEN=$(cat ~/.config/myapp/token)
DATE=$(date -u '+%Y-%m-%d')
TMPDIR=$(mktemp -d)
trap 'rm -rf "$TMPDIR"' EXIT

# Stage 1: fetch
```

```
curl -sf -H "Authorization: Bearer $TOKEN" \
  "$API_URL?date=$DATE" > "$TMPDIR/raw.json"

# Stage 2: validate and normalise (Python is best for this)
python3 <<'PY' < "$TMPDIR/raw.json" > "$TMPDIR/clean.ndjson"
import sys, json, datetime

for event in json.load(sys.stdin):
    if not event.get('timestamp') or not event.get('user_id'):
        continue
    event['ts'] = datetime.datetime.fromisoformat(
        event['timestamp']).strftime('%Y-%m-%dT%H:%M:%SZ')
    print(json.dumps(event))
PY

# Stage 3: aggregate per-user event counts (awk is ideal)
jq -r '"\(.user_id)\t\(.event_type)"' "$TMPDIR/clean.ndjson" \
  | sort \
  | awk -F'\t' '{counts[$1"/"$2]++}
    END{for(k in counts) print k, counts[k]}' \
  > "$TMPDIR/counts.txt"

# Stage 4: build final JSON report (jq)
jq -Rn '[inputs | split(" ") |
    {key: .[0], count: (.[1] | tonumber)}]' \
  "$TMPDIR/counts.txt" > "$TMPDIR/report.json"

# Stage 5: upload
curl -sf -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  --data-binary "@$TMPDIR/report.json" \
  "$API_URL/reports"

printf 'Report uploaded for %s\n' "$DATE"
```

6 — Calling Bash from Other Languages

The integration runs both ways. Sometimes your Python, Node, or Go program needs to shell out, and knowing what Bash sees on the other end helps you design the interface cleanly.

```
# ── What your script should export for callers ──────────────────
# Good script interface design:
#   - Reads config from env vars (easy for any caller)
#   - Takes structured input via stdin (JSON or line-delimited)
#   - Produces structured output on stdout
#   - Uses exit codes consistently (0 = success, non-zero = error)
#   - Writes human errors to stderr only (callers ignore stderr)

#!/usr/bin/env bash
# bin/process_order.sh  — designed to be called from Python/Node/Go
set -euo pipefail

# Config from environment
DB_HOST="${DB_HOST:?DB_HOST required}"
DB_PORT="${DB_PORT:-5432}"

# Input: JSON on stdin
order=$(cat)
order_id=$(jq -r '.id'     <<< "$order")
amount=$(  jq -r '.amount' <<< "$order")

# ... processing ...

# Output: structured JSON on stdout
jq -n --arg id "$order_id" --arg status "processed" \
  '{"order_id":$id,"status":$status}'
```

```
# ── Calling the script from Python ──────────────────────────────
import subprocess, json, os

order = {"id": "ORD-001", "amount": 99.99}

result = subprocess.run(
    ["./bin/process_order.sh"],
    input=json.dumps(order),
    capture_output=True,
    text=True,
```

```
    env={**os.environ, "DB_HOST": "localhost"},
    check=True            # raises CalledProcessError on non-zero exit
)

response = json.loads(result.stdout)
print(response["status"])   # → processed
```

```
# ── Calling from Node.js ──────────────────────────────────
const { execFile } = require('child_process');
const order = JSON.stringify({ id: 'ORD-002', amount: 49.99 });

execFile('./bin/process_order.sh', [],
  { env: { ...process.env, DB_HOST: 'localhost' } },
  (err, stdout, stderr) => {
    if (err) { console.error('Script failed:', stderr); process.exit(1); }
    const resp = JSON.parse(stdout);
    console.log(resp.status);
  }
);
```

7 — When to Stop Using Bash

The ability to hand off cleanly is as important as the ability to integrate. Continuing to grow a Bash script past its natural limits produces code that is harder to test, harder to reason about, and harder to hand to a colleague. Recognise the signals early.

Signal	What it means in practice	Language to reach for
You need real data structures (trees, graphs, objects)	Associative arrays and parallel indexed arrays no longer model the problem cleanly	Python, Go, Ruby
Error handling is more than `|| exit 1`	You need try/catch, error types, retry logic with backoff	Python, Go
You are writing >200 lines of logic	A colleague cannot understand the script without running it	Python (fastest rewrite)

You need HTTP or a database	`curl` + `jq` works up to a point; auth, retries, and pooling quickly exceed it	Python (requests/httpx), Go
You need concurrency beyond `&` + `wait`	Proper thread pools, async I/O, or actor models	Go, Python (asyncio)
You want to unit-test the logic, not the plumbing	BATS covers the shell surface; complex logic belongs in a testable unit	Python, Go, Node
Someone else will maintain this in 6 months	The safe default is Python — it is on every server and universally readable	Python
You need a CLI with subcommands, flags, help text	Writing a clean CLI in Bash is possible but `argparse` / `cobra` are far faster	Python (Click/Typer), Go (cobra)

The handoff pattern — keep the shell entry point

```bash
#!/usr/bin/env bash
# bin/run.sh — thin shell wrapper; all logic is in Python
# This is the right way to hand off: keep one entry point,
# have it exec into the real implementation.
set -euo pipefail

# Guard: require Python 3.10+
py=$(command -v python3 || { echo "python3 not found" >&2; exit 1; })
ver=$("$py" -c "import sys; print('%d%02d' % sys.version_info[:2])")
(( ver >= 310 )) || { echo "Python 3.10+ required (got $($py --version))" >&2;
exit 1; }

# Activate virtualenv if present
[[ -f "${VIRTUAL_ENV:-}/bin/activate" ]] || {
  [[ -f .venv/bin/activate ]] && source .venv/bin/activate
}

# exec replaces this shell with Python — no double-process overhead
exec "$py" -m myapp "$@"
```

8 — Practical Patterns Summary

Task	Best tool	Notes
File/process orchestration	Bash	Its strongest domain
Text transformation (simple)	awk / sed	Already in every pipeline
JSON parsing/building	jq	Use `--arg` for safe value injection
Complex text / regex	Python / Perl	Inline heredoc for short snippets
Maths beyond integer arithmetic	Python / bc	bc for simple; Python for floats/complex
HTTP requests	curl + jq	Python requests/httpx for anything non-trivial
YAML / TOML / XML	yq / python3	Never parse with grep/sed
Stateful computation in a loop	Python coprocess	One process, many requests via stdin/stdout
Parallel I/O-bound work	GNU parallel / xargs -P	Built-in parallelism without code complexity
Long-running service / daemon	Python / Go + systemd	Use shell only for the entry-point wrapper

Exercises

Exercise 1 — JSON-speaking Python coprocess

Write a complete Bash script that uses a persistent Python coprocess to perform three operations without spawning more than one Python process:

- **validate_email ADDRESS** — returns `{"ok":true}` or `{"ok":false,"reason":"..."}`
- **slugify TEXT** — lowercases, replaces spaces and non-alphanumeric characters with hyphens, collapses multiple hyphens
- **word_freq TEXT** — returns a JSON object of word → count, sorted by count descending, top 5 only

The shell script should read lines from stdin in the format `OP ARG`, call the appropriate operation via the coprocess, and print the result. Handle coprocess startup failure and include a clean shutdown on EXIT.

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

# ── Python worker ──────────────────────────────────────────
WORKER=$(mktemp --suffix=.py)
trap 'rm -f "$WORKER"' EXIT

cat > "$WORKER" <<'PY'
import sys, json, re
from collections import Counter

EMAIL_RE = re.compile(r'^[^@\s]+@[^@\s]+\.[^@\s]{2,}$')

def validate_email(addr):
    if EMAIL_RE.match(addr):
        return {"ok": True}
    return {"ok": False, "reason": "does not match email pattern"}

def slugify(text):
    s = text.lower()
    s = re.sub(r'[^a-z0-9\s-]', '', s)
    s = re.sub(r'[\s-]+', '-', s).strip('-')
    return {"ok": True, "result": s}

def word_freq(text):
    words = re.findall(r'\b\w+\b', text.lower())
    top = Counter(words).most_common(5)
    return {"ok": True, "result": {w: c for w, c in top}}

DISPATCH = {"validate_email": validate_email,
            "slugify":        slugify,
            "word_freq":      word_freq}

for raw in sys.stdin:
    raw = raw.strip()
    if not raw:
        continue
    try:
        req  = json.loads(raw)
```

```
        fn   = DISPATCH.get(req["op"])
        resp = fn(req["arg"]) if fn else {"ok": False, "reason": "unknown
op"}
    except Exception as e:
        resp = {"ok": False, "reason": str(e)}
    print(json.dumps(resp), flush=True)
PY

# ── Start coprocess ──────────────────────────────────────────
coproc PY (python3 -u "$WORKER")
[[ -n "${PY_PID:-}" ]] || { echo "Failed to start worker" >&2; exit 1; }
trap 'kill "$PY_PID" 2>/dev/null; wait "$PY_PID" 2>/dev/null' EXIT

# ── Shell dispatch ───────────────────────────────────────────
py_call() {
  local op="$1" arg="$2"
  local req
  req=$(jq -n --arg op "$op" --arg arg "$arg" '{"op":$op,"arg":$arg}')
  printf '%s\n' "$req" >&"${PY[1]}"
  local resp
  read -r resp <&"${PY[0]}"
  printf '%s\n' "$resp"
}

# ── Read OP ARG from stdin ───────────────────────────────────
while IFS= read -r line; do
  [[ -z "$line" ]] && continue
  op="${line%% *}"
  arg="${line#* }"
  py_call "$op" "$arg"
done
```

Exercise 2 — Multi-language ETL pipeline

Write a Bash script that implements a five-stage pipeline processing a CSV log file of web requests (`timestamp,method,path,status,bytes,duration_ms`):

1. **Validate** (Python inline): skip rows where status is not a 3-digit integer, duration is negative, or path contains `..`

2. **Enrich** (awk): add a sixth field `category` based on status (2xx=ok, 3xx=redirect, 4xx=client_error, 5xx=server_error)
3. **Aggregate** (awk): count requests and sum bytes by category
4. **Format** (jq): emit a JSON summary object with `date`, `total_requests`, and a `by_category` breakdown
5. **Save** (Bash): write the JSON to `reports/YYYY-MM-DD.json`, creating the directory if needed

All five stages should be connected in a single pipeline with no temporary files. Include a sample input generator at the top for testing.

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

DATE=$(date -u '+%Y-%m-%d')
REPORT_DIR="${REPORT_DIR:-reports}"

# ── Generate sample input ──────────────────────────────────
sample_csv() {
  printf 'timestamp,method,path,status,bytes,duration_ms\n'
  printf '2026-06-10T10:00:01Z,GET,/api/users,200,1234,42\n'
  printf '2026-06-10T10:00:02Z,POST,/api/orders,201,890,118\n'
  printf '2026-06-10T10:00:03Z,GET,/static/img,304,0,5\n'
  printf '2026-06-10T10:00:04Z,GET,/../etc/passwd,400,200,-1\n'  # invali
d: path + neg duration
  printf '2026-06-10T10:00:05Z,GET,/api/products,404,320,30\n'
  printf '2026-06-10T10:00:06Z,GET,/api/orders,500,450,9999\n'
  printf '2026-06-10T10:00:07Z,DELETE,/api/users/1,204,0,55\n'
  printf 'bad,row,data\n'  # invalid: too few fields
}

# ── Pipeline ──────────────────────────────────────────────
run_pipeline() {
  local input="$1"

  # Stage 1: Validate (Python) ── skip header and bad rows
  # Stage 2: Enrich (awk)      ── add category field
  # Stage 3: Aggregate (awk)   ── count + sum by category
  # Stage 4: Format (jq)       ── produce JSON

  python3 <<'PY' < "$input" \
```

```
  | awk -F, <<'AWK' \
  | awk -F, <<'AWK2' \
  | jq -Rn --arg date "$DATE" <<'JQ'
import sys, csv

r = csv.reader(sys.stdin)
next(r)  # skip header
for row in r:
    if len(row) < 6:
        continue
    ts, method, path, status, byt, dur = row[:6]
    if not status.isdigit() or len(status) != 3:
        continue
    if int(dur) < 0:
        continue
    if '..' in path:
        continue
    print(','.join(row[:6]))
PY
{
    status = int($4)
    if      (status >= 200 && status < 300) cat = "ok"
    else if (status >= 300 && status < 400) cat = "redirect"
    else if (status >= 400 && status < 500) cat = "client_error"
    else                                    cat = "server_error"
    print $0 "," cat
}
AWK
{
    cat = $7
    counts[cat]++
    bytes[cat] += $5
}
END {
    for (c in counts)
        print c "," counts[c] "," bytes[c]
}
AWK2
[ inputs | split(",") |
  { category: .[0], requests: (.[1]|tonumber), bytes: (.[2]|tonumber) } ]
|
```

```
{ date: $date,
  total_requests: (map(.requests) | add),
  by_category: (map({(.category): {requests:.requests, bytes:.bytes}}) |
add)
}
JQ
}

# ── Main ──────────────────────────────────────────────────────────
INPUT=$(mktemp)
trap 'rm -f "$INPUT"' EXIT
sample_csv > "$INPUT"

mkdir -p "$REPORT_DIR"
run_pipeline "$INPUT" > "$REPORT_DIR/$DATE.json"
printf 'Report written to %s/%s.json\n' "$REPORT_DIR" "$DATE"
cat "$REPORT_DIR/$DATE.json"
```

Exercise 3 — Bash script designed to be called from Python

Design and write a Bash script `bin/sys_info.sh` that is intended to be called from Python using `subprocess.run`. The script should:

- Accept a `--format` flag: `json` (default) or `text`
- Collect: hostname, kernel version, uptime in seconds, load averages (1/5/15 min), total/used/free memory in MB, and disk usage for `/` (total/used/free in GB)
- With `--format json`, output a single clean JSON object using `jq -n --arg/--argjson` (never string interpolation)
- With `--format text`, output a human-readable summary
- Exit 2 for unknown `--format` values
- Write all errors to stderr; keep stdout clean for machine consumption

Then write a Python snippet that calls the script, parses the JSON, and prints a one-line health summary.

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

```
FORMAT=json
while (( $# )); do
  case "$1" in
    --format) FORMAT="$2"; shift 2 ;;
    *) printf 'Unknown option: %s\n' "$1" >&2; exit 2 ;;
  esac
done

[[ $FORMAT == json || $FORMAT == text ]] || {
  printf 'Unknown format: %s (use json or text)\n' "$FORMAT" >&2; exit 2
}

# ── Collect data ──────────────────────────────────────────
HOSTNAME_VAL=$(hostname -f 2>/dev/null || hostname)
KERNEL=$(uname -r)
UPTIME_S=$(awk '{printf "%d", $1}' /proc/uptime)

# Load averages
read -r LOAD1 LOAD5 LOAD15 _rest < /proc/loadavg

# Memory in MB (from /proc/meminfo)
memval() { awk -v k="$1" '$1==k{printf "%d", int($2/1024)}' /proc/meminf
o; }
MEM_TOTAL=$(memval MemTotal:)
MEM_AVAIL=$(memval MemAvailable:)
MEM_USED=$(( MEM_TOTAL - MEM_AVAIL ))

# Disk usage for / in GB
read -r DISK_TOTAL DISK_USED DISK_FREE <<< \
  $(df -BG / | awk 'NR==2{gsub(/G/,""); print $2, $3, $4}')

# ── Output ────────────────────────────────────────────────
if [[ $FORMAT == json ]]; then
  jq -n \
    --arg     hostname   "$HOSTNAME_VAL"  \
    --arg     kernel     "$KERNEL"        \
    --argjson uptime_s   "$UPTIME_S"      \
    --argjson load1      "$LOAD1"         \
    --argjson load5      "$LOAD5"         \
    --argjson load15     "$LOAD15"        \
```

```
    --argjson mem_total  "$MEM_TOTAL"   \
    --argjson mem_used   "$MEM_USED"    \
    --argjson mem_free   "$MEM_AVAIL"   \
    --argjson disk_total "$DISK_TOTAL"  \
    --argjson disk_used  "$DISK_USED"   \
    --argjson disk_free  "$DISK_FREE"   \
    '{hostname:$hostname,kernel:$kernel,uptime_s:$uptime_s,
      load:{one:$load1,five:$load5,fifteen:$load15},
      memory:{total_mb:$mem_total,used_mb:$mem_used,free_mb:$mem_free},
      disk:{total_gb:$disk_total,used_gb:$disk_used,free_gb:$disk_free}}'
else
  printf 'Host    : %s (%s)\n'     "$HOSTNAME_VAL" "$KERNEL"
  printf 'Uptime  : %d s\n'         "$UPTIME_S"
  printf 'Load    : %s %s %s\n'     "$LOAD1" "$LOAD5" "$LOAD15"
  printf 'Memory  : %d / %d MB\n'   "$MEM_USED" "$MEM_TOTAL"
  printf 'Disk /  : %d / %d GB\n'   "$DISK_USED" "$DISK_TOTAL"
fi
```

```
# Python caller
import subprocess, json

result = subprocess.run(
    ["./bin/sys_info.sh", "--format", "json"],
    capture_output=True, text=True, check=True
)
info = json.loads(result.stdout)

mem_pct = info["memory"]["used_mb"] / info["memory"]["total_mb"] * 100
disk_pct = info["disk"]["used_gb"]  / info["disk"]["total_gb"]   * 100

print(
    f"{info['hostname']}  "
    f"load={info['load']['one']}  "
    f"mem={mem_pct:.0f}%  "
    f"disk={disk_pct:.0f}%"
)
```

Exercise 4 — The handoff: rewrite a Bash function in Python

The following Bash function has grown beyond Bash's comfortable limits. It parses a YAML-ish config file, resolves variable references (`${VAR}` within values), validates required keys, and returns results as environment variables. It is 60 lines, has two known edge-case bugs, and cannot be unit-tested without mocking the filesystem.

```
load_config() {
  local file="$1"; local -A cfg
  while IFS='=' read -r k v; do
    [[ $k =~ ^[[:space:]]*# ]] && continue
    [[ -z $k ]] && continue
    cfg["${k// /}"]="${v}"
  done < "$file"
  for k in DB_HOST DB_PORT APP_NAME; do
    [[ -v cfg[$k] ]] || { echo "missing: $k" >&2; return 1; }
  done
  for k in "${!cfg[@]}"; do
    local val="${cfg[$k]}"
    val="${val//\$\{DB_HOST\}/${cfg[DB_HOST]:-}}"
    val="${val//\$\{DB_PORT\}/${cfg[DB_PORT]:-}}"
    export "$k"="$val"
  done
}
```

Rewrite this as `bin/load_config.sh`: a thin Bash wrapper that calls a Python implementation, passes the config file path as an argument, and `eval`s the output to set the variables in the calling shell. The Python implementation should: parse the file properly (handling whitespace, inline comments, multi-word values), resolve `${VAR}` references in topological order (a variable can reference another defined earlier in the same file), validate required keys, and emit `export KEY='VALUE'` lines using `shlex.quote` so the output is safe to eval regardless of value content.

```
#!/usr/bin/env bash
# bin/load_config.sh
# Usage: eval "$(load_config.sh path/to/config)"
set -euo pipefail

CONFIG_FILE="${1:?Usage: load_config.sh CONFIG_FILE}"
[[ -f "$CONFIG_FILE" ]] || { printf 'File not found: %s\n' "$CONFIG_FILE"
>&2; exit 1; }

# Delegate entirely to Python; its stdout is safe to eval
```

```
exec python3 -c '
import sys, re, shlex

REQUIRED = {"DB_HOST", "DB_PORT", "APP_NAME"}
REF_RE   = re.compile(r"\$\{([A-Za-z_][A-Za-z0-9_]*)\}")

# ── Parse ──────────────────────────────────────────────────────
cfg    = {}   # key -> raw value (preserving order, Python 3.7+)
order  = []   # insertion order

with open(sys.argv[1]) as fh:
    for lineno, line in enumerate(fh, 1):
        line = line.rstrip("\n")
        line = re.sub(r"\s*#.*$", "", line)   # strip inline comments
        line = line.strip()
        if not line:
            continue
        if "=" not in line:
            print(f"# warning: line {lineno} skipped (no =)", file=sys.st
derr)
            continue
        k, _, v = line.partition("=")
        k = k.strip()
        v = v.strip()
        if not re.fullmatch(r"[A-Za-z_][A-Za-z0-9_]*", k):
            print(f"# warning: invalid key {k!r} on line {lineno}", file=
sys.stderr)
            continue
        cfg[k] = v
        if k not in order:
            order.append(k)

# ── Validate ───────────────────────────────────────────────────
missing = REQUIRED - set(cfg)
if missing:
    for m in sorted(missing):
        print(f"echo Missing required key: {m} >&2", file=sys.stdout)
    print("exit 1")
    sys.exit(0)

# ── Resolve variable references (single pass in definition order) —
```

```
resolved = {}
for k in order:
    def replacer(m, _r=resolved):
        return _r.get(m.group(1), m.group(0))
    resolved[k] = REF_RE.sub(replacer, cfg[k])

# ── Emit safe export statements ──────────────────────────
for k in order:
    print(f"export {k}={shlex.quote(resolved[k])}")
' -- "$CONFIG_FILE"
```

```
# Example usage in another script:
#   eval "$(bin/load_config.sh config/app.conf)"
#   echo "Connecting to $DB_HOST:$DB_PORT"
#
# Example config/app.conf:
#   DB_HOST = db.internal       # primary database
#   DB_PORT = 5432
#   APP_NAME = my-service
#   DB_URL = postgres://${DB_HOST}:${DB_PORT}/mydb
#   GREETING = Hello from ${APP_NAME}
```