A VISUAL REFERENCE · Bash

Bash, one change at a time

From a first command to small, reliable shell programs. Complete source. A small change. Its actual result.

The Org file is the canonical source. Read the HTML guide beside your work, or download its PDF and source for offline reading.

> **Last changes: 2026-10-19.** Added a searchable HTML gallery with code-and-output previews, labelled Zsh comparisons and links to complete examples and back. Every tour now links to tours.emaix.org. Unified tour–chapter–example IDs, numbered chapter anchors and updated direct links, downloads and comparisons. Added nine worked examples for here-strings, variable-name expansion and ERR traps, with an expanded quick reference and a Devhints cheatsheet link. Reader pages now omit publishing instructions and highlight PDF downloads.

152
worked examples

21
topic families

1
canonical Org source

READING THE TOUR

Three ways to see a change

Program

Read a complete script with its own input and an explicit interpreter.

Captured result

Compare standard output, standard error and exit status with the code.

One change

Follow stable identifiers and focused explanations without an accumulating shell session.

Bash is the main interpreter. Amber Zsh difference notes are optional comparisons. They are explicitly labelled; a Bash block requires Bash even when your interactive terminal normally starts Zsh.

Read from the beginning or jump to a stable example ID. Each example includes its complete input and captured result. Copy link preserves the address of the guide you are reading, including its current domain and path.

Bash quick reference

Use these reminders to find a complete program and its captured result. Each example link opens the corresponding section of this tour, including offline. For another compact reference, see the Devhints Bash scripting cheatsheet.

Topic	Reminder	Worked example
Quoting	Keep a scalar together with `"$value"`.	BSH-02-003
Defaults	Use `${value:-fallback}` for an unset or empty value.	BSH-03-001
Path suffixes	Remove a matching suffix with `${file%.*}`.	BSH-03-004
Substrings	Extract a slice with `${text:offset:length}`.	BSH-17-001
Script arguments	Preserve argument boundaries with `"$@"`.	BSH-07-004
Arrays	Keep elements separate with `"${items[@]}"`.	BSH-08-003
Reading lines	Use `IFS= read -r line` to preserve spaces and backslashes.	BSH-07-002
Pipeline failures	Enable `set -o pipefail` to expose an earlier failure.	BSH-06-008
Function status	Test success explicitly with `if task; then`.	BSH-21-005
Process substitution	Read command output through `<(command)`.	BSH-09-005
Regular expressions	Read captures from `BASH_REMATCH` after a successful match.	BSH-17-006
Short options	Parse flags and their values with `getopts`.	BSH-12-002
Here-strings	Feed a string to standard input with `<<< "$text"`.	BSH-05-012
Variable names	Discover names with `"${!tour_demo_@}"`.	BSH-19-007
Error traps	Report an unchecked failure with `trap 'handler' ERR`.	BSH-16-007

The reminders describe Bash. Follow the labelled Zsh comparisons in the worked examples when using Zsh; array indexing and expansion rules can differ.

Read this first

This is a Bash reference, not an exercise book. The main examples use Bash; amber Zsh difference notes are optional comparisons, not required steps. Read each small program together with its actual output. Stable BSH-CC-NNN identifiers survive changes to titles and page order. Each example includes its own input data and runs independently.

The Org source contains each complete program. Read its captured stdout, stderr and exit status beside the code. Bash is the main interpreter; explicitly marked Zsh difference notes show alternative syntax or behavior.

Execution profile: noninteractive Bash --noprofile --norc, or native Zsh -f; LC_ALL=C; fresh temporary HOME. Zsh may still read a system zshenv. Interactive configuration and plugins are outside the test profile. The examples use shell builtins and common Unix utilities. External tool behavior can differ on macOS/BSD; tested versions are recorded in the build report. A Bash block always requires Bash even if your terminal normally starts Zsh.

The build isolates working files but is not a security sandbox. Review code before executing it. In Emacs, load ob-shell (M-x load-library RET ob-shell), then use C-c C-c on a bash or zsh block. Run file-writing blocks in a disposable directory: unlike the supplied builder, plain Babel execution does not create an isolated directory automatically. :results output captures stdout; the custom builder additionally records stderr and exit status.

Interactive companion

These actions require a real terminal and are not part of the executable tests. Tab requests completion; Ctrl-R searches history in common Emacs-style keymaps; Ctrl-A and Ctrl-E move to the start and end of the editing line. Bindings depend on Readline (Bash), ZLE (Zsh), vi/emacs mode and personal configuration.

Use type printf to inspect command resolution. Use bash --version or zsh --version to check the actual interpreter; $SHELL usually describes the login shell, not necessarily the running interpreter. Use bash script.sh or zsh script.zsh explicitly. For an executable script, add #!/usr/bin/env bash or #!/usr/bin/env zsh as its first line, then chmod +x script.sh and run ./script.sh. A shebang is ignored when you explicitly invoke a different interpreter.

Bash interactive configuration commonly belongs in ~/.bashrc; interactive Zsh configuration in ~/.zshrc. Login startup differs. Do not put tour experiments into startup files. Completion setup and prompt themes are outside this scripting tour.

Sources and scope

The explanations were checked against the official Bash Reference Manual (quoting, expansions, arrays, pipelines), the official Zsh manual (expansion, parameters and options) and the Org Babel shell documentation. Output panels are generated by running the supplied code, not copied from those sources.

- GNU Bash Reference Manual
- Zsh manual
- Org Babel shell documentation
- Devhints Bash scripting cheatsheet (additional quick reference)

Do not treat set -e as universal error handling: its behavior depends on command context. Prefer explicit validation and deliberate status handling; pipefail helps expose pipeline failures. Never parse ls output to obtain filenames. Arrays and quoted expansions preserve argument boundaries. This expanded tour covers scripting fundamentals and selected advanced patterns. Interactive job control, completion systems and terminal styling remain outside the executable reference.

First commands and formatted output

BSH-01-001

Print a line

printf is a shell builtin. %s inserts a string; \n ends the line. Keep the format string constant.

Code

```
printf '%s\n' 'Hello, shell!'
```

Actual output

bash · exit 0

```
Hello, shell!
```

zsh · exit 0

```
Hello, shell!
```

BSH-01-002

One argument, one line

Change: supply two arguments. printf reuses the format for remaining arguments.

Code

```
printf '%s\n' 'alpha' 'beta'
```

Actual output

bash · exit 0

```
alpha
beta
```

zsh · exit 0

```
alpha
beta
```

BSH-01-003

Keep words together

Change: spaces inside quotes remain part of one argument. Angle brackets expose argument boundaries.

Code

```
printf '<%s>\n' 'alpha beta' gamma
```

Actual output

bash · exit 0

```
<alpha beta>
<gamma>
```

zsh · exit 0

```
<alpha beta>
<gamma>
```

BSH-01-004

Align columns

%8s right-aligns in eight characters; %-8s left-aligns. %04d pads an integer with zeros.

Code

```
printf '%-8s %04d\n' apples 7 pears 12
```

Actual output

bash · exit 0

```
apples   0007
pears    0012
```

zsh · exit 0

```
apples   0007
pears    0012
```

BSH-01-005

Choose decimal precision

%.2f prints two digits after the decimal point. Locale is fixed to C throughout this tour.

Code

```
printf 'mean = %.2f\n' 3.14159
```

Actual output

bash · exit 0

```
mean = 3.14
```

zsh · exit 0

```
mean = 3.14
```

BSH-01-006

Comments and continuation

A backslash immediately before a newline continues a command. Do not put spaces after it.

Code

```
# Two physical lines, one command
printf '%s\n' \
  'continued command'
```

Actual output

bash · exit 0

```
continued command
```

zsh · exit 0

```
continued command
```

Variables and quoting

BSH-02-001

Assign and expand

No spaces around = in an assignment. Double quotes expand the variable while preserving one argument.

Code

```
label='sample A'
printf '<%s>\n' "$label"
```

Actual output

bash · exit 0

```
<sample A>
```

zsh · exit 0

```
<sample A>
```

BSH-02-002

Single versus double quotes

Change: single quotes preserve the dollar sign literally; double quotes expand the variable.

Code

```
label='sample A'
printf '%s\n' '$label' "$label"
```

Actual output

bash · exit 0

```
$label
sample A
```

zsh · exit 0

```
$label
sample A
```

BSH-02-003

Bash splits an unquoted scalar

Change: remove double quotes. Default Bash splits on IFS whitespace. Use quoted expansions for filenames.

Code

```
label='sample A'
printf '<%s>\n' $label
```

Actual output

bash · exit 0

```
<sample>
<A>
```

Zsh difference: keeps an unquoted scalar

Same code, different shell: native Zsh does not normally split scalar expansions. Quoting is still a good shared convention.

Code

```
label='sample A'
printf '<%s>\n' $label
```

Actual output

zsh · exit 0

```
<sample A>
```

Zsh difference: Explicit Zsh splitting

Change: ${=label} explicitly requests word splitting in Zsh.

Code

```
label='sample A'
printf '<%s>\n' ${=label}
```

Actual output

zsh · exit 0

```
<sample>
<A>
```

BSH-02-006

Protect a variable boundary

Braces delimit the name. Without braces, $label_final would name a different variable.

Code

```
label=sample
printf '%s\n' "${label}_final"
```

Actual output

bash · exit 0

```
sample_final
```

zsh · exit 0

```
sample_final
```

Parameter expansion and arithmetic

BSH-03-001

A fallback value

${x:-word} uses word when x is unset or empty. It does not assign x.

Code

```
unset label
printf '%s\n' "${label:-untitled}"
```

Actual output

bash · exit 0

```
untitled
```

zsh · exit 0

```
untitled
```

BSH-03-002

Unset versus empty

Change: remove the colon. ${x-word} substitutes only when x is unset, not when it is an empty string.

Code

```
label=''
printf 'colon=<%s> plain=<%s>\n' "${label:-fallback}" "${label-fallback}"
```

Actual output

bash · exit 0

```
colon=<fallback> plain=<>
```

zsh · exit 0

```
colon=<fallback> plain=<>
```

BSH-03-003

Assign the fallback

Change: := also assigns the default. Later uses see the assigned value.

Code

```
unset label
printf '%s\n' "${label:=untitled}" "$label"
```

Actual output

bash · exit 0

```
untitled
untitled
```

zsh · exit 0

```
untitled
untitled
```

BSH-03-004

Remove a suffix

% removes the shortest matching suffix. Here .csv is a literal suffix.

Code

```
file='sample.raw.csv'
printf '%s\n' "${file%.csv}"
```

Actual output

bash · exit 0

```
sample.raw
```

zsh · exit 0

```
sample.raw
```

BSH-03-005

Shortest versus longest match

% removes the shortest matching suffix; %% removes the longest. * is a shell pattern here.

Code

```
file='sample.raw.csv'
printf '%s\n' "${file%.*}" "${file%%.*}"
```

Actual output

bash · exit 0

```
sample.raw
sample
```

zsh · exit 0

```
sample.raw
sample
```

BSH-03-006

Remove a path prefix

removes the longest matching prefix. This is a string operation, not a filesystem lookup.

Code

```
file='data/raw/sample.csv'
printf '%s\n' "${file##*/}"
```

Actual output

bash · exit 0

```
sample.csv
```

zsh · exit 0

```
sample.csv
```

BSH-03-007

Replace matching text

/ replaces the first match; // replaces all matches. These are patterns, not regular expressions.

Code

```
label='a-b-c'
printf '%s\n' "${label/-/_}" "${label//-/_}"
```

Actual output

bash · exit 0

```
a_b-c
a_b_c
```

zsh · exit 0

```
a_b-c
a_b_c
```

BSH-03-008

Integer arithmetic

$((...)) expands an integer result. Integer division truncates toward zero.

Code

```
n=7
printf '%s\n' "$((n + 2))" "$((n / 2))" "$((n % 2))"
```

Actual output

bash · exit 0

```
9
3
1
```

zsh · exit 0

```
9
3
1
```

Files and filename patterns

BSH-04-001

Create, copy and read

mkdir creates a directory; cp copies a file; cat writes its contents. All files are local to the example sandbox.

Code

```
mkdir data
printf '%s\n' alpha > data/input.txt
cp data/input.txt data/copy.txt
cat data/copy.txt
```

Actual output

bash · exit 0

```
alpha
```

zsh · exit 0

```
alpha
```

BSH-04-002

Select a filename suffix

The unquoted *.txt is expanded by the shell before printf runs. Results use C-locale ordering.

Code

```
touch a.txt b.txt c.csv
printf '%s\n' *.txt
```

Actual output

bash · exit 0

```
a.txt
b.txt
```

zsh · exit 0

```
a.txt
b.txt
```

BSH-04-003

A quoted pattern stays literal

Change: quote the pattern. The shell now passes a literal asterisk.

Code

```
touch a.txt b.txt c.csv
printf '%s\n' '*.txt'
```

Actual output

bash · exit 0

```
*.txt
```

zsh · exit 0

```
*.txt
```

BSH-04-004

Exactly one character

? matches one character; [ab] matches one character from a set.

Code

```
touch a1.txt a2.txt b1.txt aa1.txt
printf '%s\n' [ab]?.txt
```

Actual output

bash · exit 0

```
a1.txt
a2.txt
b1.txt
```

zsh · exit 0

```
a1.txt
a2.txt
b1.txt
```

BSH-04-005

Brace expansion does not find files

Braces generate words even if no matching files exist. This feature is shared by Bash and Zsh, not all sh implementations.

Code

```
printf '%s\n' sample_{A,B}.csv
```

Actual output

bash · exit 0

```
sample_A.csv
sample_B.csv
```

zsh · exit 0

```
sample_A.csv
sample_B.csv
```

BSH-04-006

Bash unmatched glob

Default Bash retains an unmatched pattern as literal text.

Code

```
printf '<%s>\n' missing*.csv
```

Actual output

bash · exit 0

```
<missing*.csv>
```

Zsh difference: unmatched glob

Default native Zsh reports an error and does not run printf. This example intentionally exits nonzero.

Code

```
printf '<%s>\n' missing*.csv
```

Actual output

zsh · exit 1

```
(no stdout)
```

Standard error

```
zsh:1: no matches found: missing*.csv
```

BSH-04-008

Bash: collect zero matches

nullglob removes unmatched patterns. An array then accurately contains zero filenames.

Code

```
shopt -s nullglob
files=(missing*.csv)
printf 'files=%s\n' "${#files[@]}"
```

Actual output

bash · exit 0

```
files=0
```

Zsh difference: collect zero matches

The (N) glob qualifier removes this pattern when it has no matches; it does not change all future globs.

Code

```
files=(missing*.csv(N))
printf 'files=%s\n' "${#files[@]}"
```

Actual output

zsh · exit 0

```
files=0
```

Streams, redirection and pipes

BSH-05-001

Write a file

> redirects stdout and replaces existing contents. cat then displays the file.

Code

```
printf '%s\n' alpha > notes.txt
cat notes.txt
```

Actual output

bash · exit 0

```
alpha
```

zsh · exit 0

```
alpha
```

BSH-05-002

Append instead

Change: >> appends rather than replacing.

Code

```
printf '%s\n' alpha > notes.txt
printf '%s\n' beta >> notes.txt
cat notes.txt
```

Actual output

bash · exit 0

```
alpha
beta
```

zsh · exit 0

```
alpha
beta
```

BSH-05-003

Read standard input

< connects a file to stdin. tr replaces lowercase letters with uppercase letters.

Code

```
printf '%s\n' alpha > notes.txt
tr 'a-z' 'A-Z' < notes.txt
```

Actual output

bash · exit 0

```
ALPHA
```

zsh · exit 0

```
ALPHA
```

BSH-05-004

Connect two commands

| connects stdout of the left command to stdin of the right command. It does not pipe stderr by default. |

Code

```
printf '%s\n' alpha beta | tr 'a-z' 'A-Z'
```

Actual output

bash · exit 0

```
ALPHA
BETA
```

zsh · exit 0

```
ALPHA
BETA
```

BSH-05-005

Sort and count repeats

sort groups equal adjacent lines; uniq -c counts each group. Both are external utilities.

Code

```
printf '%s\n' pear apple pear | sort | uniq -c
```

Actual output

bash · exit 0

```
1 apple
2 pear
```

zsh · exit 0

```
1 apple
2 pear
```

BSH-05-006

Separate stdout and stderr

File descriptor 1 is stdout; 2 is stderr. >&2 sends this printf to stderr.

Code

```
printf '%s\n' result
printf '%s\n' diagnostic >&2
```

Actual output

bash · exit 0

```
result
```

Standard error

```
diagnostic
```

zsh · exit 0

```
result
```

Standard error

```
diagnostic
```

BSH-05-007

Capture both streams

> captures stdout, then 2>&1 points stderr at that destination. The braces group commands in the current shell.

Code

```
{ printf 'result\n'; printf 'diagnostic\n' >&2; } > log.txt 2>&1
cat log.txt
```

Actual output

bash · exit 0

```
result
diagnostic
```

zsh · exit 0

```
result
diagnostic
```

BSH-05-008

Redirection order matters

Change: reverse the redirections. stderr keeps the original stdout destination while stdout goes to the file.

Code

```
{ printf 'result\n'; printf 'diagnostic\n' >&2; } 2>&1 > log.txt
printf 'FILE:\n'
cat log.txt
```

Actual output

bash · exit 0

```
diagnostic
FILE:
result
```

zsh · exit 0

```
diagnostic
FILE:
result
```

BSH-05-009

A literal here-document

Quoting the delimiter prevents expansion in the document. The closing delimiter starts at column one.

Code

```
label=sample
cat <<'TEXT'
$label is literal here.
TEXT
```

Actual output

bash · exit 0

```
$label is literal here.
```

zsh · exit 0

```
$label is literal here.
```

BSH-05-010

An expanding here-document

Change: leave the delimiter unquoted. Variable and command substitutions are now expanded.

Code

```
label=sample
cat <<TEXT
$label is expanded here.
TEXT
```

Actual output

bash · exit 0

```
sample is expanded here.
```

zsh · exit 0

```
sample is expanded here.
```

BSH-05-011

Save and continue a stream

tee writes a copy to a file and passes the same stream onward. The second command reads the saved copy.

Code

```
printf 'alpha\n' | tee copy.txt | tr 'a-z' 'A-Z'
cat copy.txt
```

Actual output

bash · exit 0

```
ALPHA
alpha
```

zsh · exit 0

```
ALPHA
alpha
```

BSH-05-012

Feed a variable through a here-string

<<< sends one expanded string to standard input and appends a newline. Double quotes make the intended input clear; no temporary file or producer pipeline is needed.

Code

```
text='alpha beta'
tr '[:lower:]' '[:upper:]' <<< "$text"
```

Actual output

bash · exit 0

```
ALPHA BETA
```

BSH-05-013

An empty here-string still supplies a line

Change: use an empty string. The here-string still supplies its appended newline, so read succeeds with an empty line. printf without a newline supplies no bytes, so the second read reaches EOF and returns 1. The group prints that status without treating EOF as a successful read.

Code

```
text=''
IFS= read -r line <<< "$text"
printf 'here-string: status=%s length=%s\n' "$?" "${#line}"
printf '%s' "$text" | {
  IFS= read -r line
  printf 'printf: status=%s length=%s\n' "$?" "${#line}"
}
```

Actual output

bash · exit 0

```
here-string: status=0 length=0
printf: status=1 length=0
```

Exit status and decisions

BSH-06-001

Read the exit status

$? is the most recent command status. Zero means success; nonzero has command-specific meaning.

Code

```
false
printf 'status=%s\n' "$?"
```

Actual output

bash · exit 0

```
status=1
```

zsh · exit 0

```
status=1
```

BSH-06-002

Run only after success

&& runs its right-hand command only if the left-hand command succeeds.

Code

```
true && printf 'runs\n'
false && printf 'skipped\n'
printf 'done\n'
```

Actual output

bash · exit 0

```
runs
done
```

zsh · exit 0

```
runs
done
```

BSH-06-003

Recover after failure

| | runs its right-hand command only if the left-hand command fails. |

Code

```
false || printf 'fallback\n'
```

Actual output

bash · exit 0

```
fallback
```

zsh · exit 0

```
fallback
```

BSH-06-004

Branch on a number

[[...]] is a Bash/Zsh conditional construct; -gt means numerically greater than.

Code

```
n=7
if [[ "$n" -gt 5 ]]; then
  printf 'large\n'
else
  printf 'small\n'
fi
```

Actual output

bash · exit 0

```
large
```

zsh · exit 0

```
large
```

BSH-06-005

Match a string pattern

An unquoted pattern on the right of == performs pattern matching inside [[...]].

Code

```
file='sample.csv'
if [[ "$file" == *.csv ]]; then
  printf 'CSV input\n'
fi
```

Actual output

bash · exit 0

```
CSV input
```

zsh · exit 0

```
CSV input
```

BSH-06-006

Several alternatives with case

Patterns before) select a branch; ;; ends a branch; * provides a catch-all.

Code

```
file='sample.tsv'
case "$file" in
  *.csv) printf 'comma\n' ;;
  *.tsv) printf 'tab\n' ;;
  *) printf 'unknown\n' ;;
esac
```

Actual output

bash · exit 0

```
tab
```

zsh · exit 0

```
tab
```

BSH-06-007

Default pipeline status

Without pipefail, the final command determines the pipeline status.

Code

```
false | cat
printf 'pipeline=%s\n' "$?"
```

Actual output

bash · exit 0

```
pipeline=0
```

zsh · exit 0

```
pipeline=0
```

BSH-06-008

Enable pipefail

Change: pipefail makes the rightmost failing command determine status, or zero when all succeed.

Code

```
set -o pipefail
false | cat
printf 'pipeline=%s\n' "$?"
```

Actual output

bash · exit 0

```
pipeline=1
```

zsh · exit 0

```
pipeline=1
```

Loops, arguments and functions

BSH-07-001

Loop over explicit arguments

for assigns each argument to item, then runs the body. Quoted words remain single items.

Code

```
for item in alpha 'beta gamma'; do
  printf '<%s>\n' "$item"
done
```

Actual output

bash · exit 0

```
<alpha>
<beta gamma>
```

zsh · exit 0

```
<alpha>
<beta gamma>
```

BSH-07-002

Read complete lines

IFS= avoids trimming; read -r preserves backslashes. Input redirection avoids a pipeline subshell.

Code

```
printf '%s\n' '  alpha' 'a\b' > lines.txt
while IFS= read -r line; do
  printf '<%s>\n' "$line"
done < lines.txt
```

Actual output

bash · exit 0

```
<  alpha>
<a\b>
```

zsh · exit 0

```
<  alpha>
<a\b>
```

BSH-07-003

Repeat while a condition holds

-lt means less than. Assignment with arithmetic expansion increments the counter.

Code

```
n=0
while [[ "$n" -lt 3 ]]; do
  printf '%s\n' "$n"
  n=$((n + 1))
done
```

Actual output

bash · exit 0

```
0
1
2
```

zsh · exit 0

```
0
1
2
```

BSH-07-004

Preserve positional arguments

set -- sets this shell's arguments. "$@" preserves each argument, including spaces.

Code

```
set -- alpha 'beta gamma'
printf 'count=%s\n' "$#"
printf '<%s>\n' "$@"
```

Actual output

bash · exit 0

```
count=2
<alpha>
<beta gamma>
```

zsh · exit 0

```
count=2
<alpha>
<beta gamma>
```

BSH-07-005

Join positional arguments

Change: "$*" forms one argument, joined by the first character of IFS (normally a space).

Code

```
set -- alpha 'beta gamma'
printf '<%s>\n' "$*"
```

Actual output

bash · exit 0

```
<alpha beta gamma>
```

zsh · exit 0

```
<alpha beta gamma>
```

BSH-07-006

A function with an argument

Functions receive their own positional arguments. "$1" is the first; local limits a variable to function scope.

Code

```
greet() {
  local person="$1"
  printf 'Hello, %s!\n' "$person"
}
greet 'Ada Lovelace'
```

Actual output

bash · exit 0

```
Hello, Ada Lovelace!
```

zsh · exit 0

```
Hello, Ada Lovelace!
```

BSH-07-007

Capture a command result

$(...) captures stdout and removes trailing newlines. Quote the result when expanding it.

Code

```
greet() { printf 'Hello, %s!\n' "$1"; }
message=$(greet Ada)
printf '<%s>\n' "$message"
```

Actual output

bash · exit 0

```
<Hello, Ada!>
```

zsh · exit 0

```
<Hello, Ada!>
```

BSH-07-008

Return a status from a function

return sets an exit status, not a text value. Use stdout for data and status for success/failure.

Code

```
is_csv() { [[ "$1" == *.csv ]]; }
if is_csv sample.csv; then printf 'accepted\n'; fi
```

Actual output

bash · exit 0

```
accepted
```

zsh · exit 0

```
accepted
```

Arrays and mappings

BSH-08-001

Bash starts at index zero

Indexed Bash arrays normally start at 0. Use braces around indexed expansions.

Code

```
items=(alpha 'beta gamma')
printf '%s\n' "${items[0]}" "${items[1]}"
```

Actual output

bash · exit 0

```
alpha
beta gamma
```

Zsh difference: starts at index one

Native Zsh arrays normally start at 1. This assumes KSH_ARRAYS is not enabled.

Code

```
items=(alpha 'beta gamma')
printf '%s\n' "${items[1]}" "${items[2]}"
```

Actual output

zsh · exit 0

```
alpha
beta gamma
```

BSH-08-003

Preserve array elements

"${items[@]}" passes each element separately in both shells, including elements containing spaces.

Code

```
items=(alpha 'beta gamma')
printf 'count=%s\n' "${#items[@]}"
printf '<%s>\n' "${items[@]}"
```

Actual output

bash · exit 0

```
count=2
<alpha>
<beta gamma>
```

zsh · exit 0

```
count=2
<alpha>
<beta gamma>
```

BSH-08-004

Append an array element

+= appends elements to an indexed array.

Code

```
items=(alpha)
items+=('beta gamma')
printf '<%s>\n' "${items[@]}"
```

Actual output

bash · exit 0

```
<alpha>
<beta gamma>
```

zsh · exit 0

```
<alpha>
<beta gamma>
```

BSH-08-005

Bash associative array

declare -A requires Bash 4 or newer. A system Bash 3.2 cannot run this example.

Code

```
declare -A units
units[mass]=kg
units[time]=s
printf '%s\n' "${units[mass]}"
```

Actual output

bash · exit 0

```
kg
```

Zsh difference: associative array

Change: declare the mapping with typeset -A in Zsh.

Code

```
typeset -A units
units[mass]=kg
units[time]=s
printf '%s\n' "${units[mass]}"
```

Actual output

zsh · exit 0

```
kg
```

Zsh difference: split and join flags

(s:,:) splits on a comma; (j: / :) joins the resulting array with a chosen separator.

Code

```
text='alpha,beta,gamma'
items=("${(@s:,:)text}")
printf '%s\n' "${(j: / :)items}"
```

Actual output

zsh · exit 0

```
alpha / beta / gamma
```

Zsh difference: path modifiers

:t selects the tail (basename); :h the directory; :r removes the final extension. No external basename command is needed.

Code

```
file='data/sample.csv'
printf '%s\n' "${file:t}" "${file:h}" "${file:r}"
```

Actual output

zsh · exit 0

```
sample.csv
data
data/sample
```

Processes, isolation and robust scripts

BSH-09-001

A subshell isolates assignments

(...) runs in a subshell environment. Changes to n do not propagate back.

Code

```
n=1
(n=2; printf 'inside=%s\n' "$n")
printf 'outside=%s\n' "$n"
```

Actual output

bash · exit 0

```
inside=2
outside=1
```

zsh · exit 0

```
inside=2
outside=1
```

BSH-09-002

A group shares assignments

Change: {...;} groups commands in the current shell; the assignment persists.

Code

```
n=1
{ n=2; printf 'inside=%s\n' "$n"; }
printf 'outside=%s\n' "$n"
```

Actual output

bash · exit 0

```
inside=2
outside=2
```

zsh · exit 0

```
inside=2
outside=2
```

BSH-09-003

Export to a child process

export adds a variable to the environment of future child processes. The child here is explicitly Bash.

Code

```
export TOUR_LABEL=sample
bash --noprofile --norc -c 'printf "%s\n" "$TOUR_LABEL"'
```

Actual output

bash · exit 0

```
sample
```

zsh · exit 0

```
sample
```

BSH-09-004

Wait for a background job

& starts a job asynchronously. $! identifies the last background job; wait retrieves its exit status.

Code

```
(exit 7) &
job_pid=$!
wait "$job_pid"
printf 'job status=%s\n' "$?"
```

Actual output

bash · exit 0

```
job status=7
```

zsh · exit 0

```
job status=7
```

Standard error

```
zsh:1: nice(5) failed: operation not permitted
```

BSH-09-005

Process substitution

<(...) supplies a path from which a command can read generated output. diff compares the two sorted streams.

Code

```
diff <(printf 'b\na\n' | sort) <(printf 'a\nb\n')
printf 'diff status=%s\n' "$?"
```

Actual output

bash · exit 0

```
diff status=0
```

zsh · exit 0

```
diff status=0
```

BSH-09-006

Unset variables become errors

set -u rejects an unset variable expansion. This example intentionally fails; it does not demonstrate all errexit rules.

Code

```
set -u
unset missing
printf '%s\n' "$missing"
```

Actual output

bash · exit 127

```
(no stdout)
```

Standard error

```
bash: line 3: missing: unbound variable
```

zsh · exit 1

```
(no stdout)
```

Standard error

```
zsh:3: missing: parameter not set
```

BSH-09-007

A fallback works with nounset

Change: supply a default for the potentially unset variable.

Code

```
set -u
unset missing
printf '%s\n' "${missing:-fallback}"
```

Actual output

bash · exit 0

```
fallback
```

zsh · exit 0

```
fallback
```

BSH-09-008

Cleanup on shell exit

trap registers an exit action. Only the file created by this example is removed.

Code

```
printf 'temporary\n' > scratch.txt
trap 'rm -f scratch.txt; printf "cleaned up\n"' EXIT
cat scratch.txt
```

Actual output

bash · exit 0

```
temporary
cleaned up
```

zsh · exit 0

```
temporary
cleaned up
```

A small reproducible report

BSH-10-001

Stage a tiny dataset

A quoted here-document creates a predictable input file. This is TSV data with actual tab characters.

Code

```
cat > measurements.tsv <<'DATA'
sample  value
A       3
B       7
C       5
DATA
cat measurements.tsv
```

Actual output

bash · exit 0

```
sample  value
A       3
B       7
C       5
```

zsh · exit 0

```
sample  value
A       3
B       7
C       5
```

BSH-10-002

Validate before processing

A readable nonempty file passes -r and -s. Explicit checks give a meaningful failure message.

Code

```
printf 'sample\tvalue\nA\t3\n' > measurements.tsv
if [[ ! -r measurements.tsv || ! -s measurements.tsv ]]; then
  printf 'missing input\n' >&2
  exit 1
fi
printf 'input accepted\n'
```

Actual output

bash · exit 0

```
input accepted
```

zsh · exit 0

```
input accepted
```

BSH-10-003

Compute and write a report

awk is an external tool: NR>1 skips the header; sum/count yields the mean. Every fixture is repeated so this example stands alone.

Code

```
printf 'sample\tvalue\nA\t3\nB\t7\nC\t5\n' > measurements.tsv
awk 'NR>1 {sum+=$2; n++}
     END {printf "samples=%d\nmean=%.2f\n", n, sum/n}' \
  measurements.tsv > report.txt
cat report.txt
```

Actual output

bash · exit 0

```
samples=3
mean=5.00
```

zsh · exit 0

```
samples=3
mean=5.00
```

Filename-safe processing

BSH-11-001

Keep a space inside a filename

An array stores filenames as separate elements. Quoting the array expansion preserves spaces without parsing directory listings.

Code

```
touch 'sample A.txt' 'sample B.txt'
files=(*.txt)
for file in "${files[@]}"; do
  printf '<%s>\n' "$file"
done
```

Actual output

bash · exit 0

```
<sample A.txt>
<sample B.txt>
```

zsh · exit 0

```
<sample A.txt>
<sample B.txt>
```

BSH-11-002

A leading dash is still a filename

Prefixing a relative name with ./ prevents cat from treating -n as an option. This convention also works with utilities that lack --.

Code

```
printf 'contents\n' > ./-n
cat ./-n
```

Actual output

bash · exit 0

```
contents
```

zsh · exit 0

```
contents
```

BSH-11-003

Read NUL-separated records

read -d sets the delimiter; an empty delimiter means NUL here. NUL can separate names containing spaces or newlines. Shell variables cannot store NUL bytes.

Code

```
printf '%s\0' 'sample A.txt' 'sample B.txt' > names.bin
while IFS= read -r -d '' file; do
  printf '<%s>\n' "$file"
done < names.bin
```

Actual output

bash · exit 0

```
<sample A.txt>
<sample B.txt>
```

zsh · exit 0

```
<sample A.txt>
<sample B.txt>
```

BSH-11-004

Process the last unterminated line

Change: the || test retains the final nonempty record even when read reaches EOF without a newline.

Code

```
printf 'alpha\nbeta' > lines.txt
while IFS= read -r line || [[ -n "$line" ]]; do
  printf '<%s>\n' "$line"
done < lines.txt
```

Actual output

bash · exit 0

```
<alpha>
<beta>
```

zsh · exit 0

```
<alpha>
<beta>
```

BSH-11-005

Read two fields

IFS=: splits a simple colon-delimited line. The last variable receives the remaining fields. This is not a general CSV parser.

Code

```
printf 'sample:7:kg\n' > data.txt
while IFS=: read -r name rest; do
  printf 'name=%s rest=%s\n' "$name" "$rest"
done < data.txt
```

Actual output

bash · exit 0

```
name=sample rest=7:kg
```

zsh · exit 0

```
name=sample rest=7:kg
```

BSH-11-006

Skip a file deliberately

continue skips the rest of the current iteration. The pattern is a shell pattern, not a regular expression.

Code

```
for file in a.csv notes.txt b.csv; do
  [[ "$file" == *.csv ]] || continue
  printf 'process %s\n' "$file"
done
```

Actual output

bash · exit 0

```
process a.csv
process b.csv
```

zsh · exit 0

```
process a.csv
process b.csv
```

Script options one step at a time

BSH-12-001

Consume one argument

shift removes the first positional argument and renumbers the rest. Check the argument count before shifting in real scripts.

Code

```
set -- alpha beta gamma
printf 'before=%s\n' "$1"
shift
printf 'after=%s count=%s\n' "$1" "$#"
```

Actual output

bash · exit 0

```
before=alpha
after=beta count=2
```

zsh · exit 0

```
before=alpha
after=beta count=2
```

BSH-12-002

Parse a short flag

getopts scans short options; opt receives the option letter. OPTIND=1 starts a fresh scan. The leading colon selects silent error reporting.

Code

```
set -- -v
OPTIND=1
verbose=no
while getopts ':v' opt; do
  case "$opt" in v) verbose=yes ;; esac
done
printf 'verbose=%s\n' "$verbose"
```

Actual output

bash · exit 0

```
verbose=yes
```

zsh · exit 0

```
verbose=yes
```

BSH-12-003

An option with a value

Change: n: means -n requires an argument. OPTARG receives it; quotes preserve spaces.

Code

```
set -- -n 'sample A'
OPTIND=1
while getopts ':n:' opt; do
  case "$opt" in n) printf 'name=%s\n' "$OPTARG" ;; esac
done
```

Actual output

bash · exit 0

```
name=sample A
```

zsh · exit 0

```
name=sample A
```

BSH-12-004

Keep the remaining filenames

After option parsing, shift by OPTIND-1 removes consumed arguments. -- marks the end of options.

Code

```
set -- -n demo -- 'sample A.csv' b.csv
OPTIND=1
while getopts ':n:' opt; do
  case "$opt" in n) printf 'name=%s\n' "$OPTARG" ;; esac
done
shift "$((OPTIND - 1))"
printf 'file=<%s>\n' "$@"
```

Actual output

bash · exit 0

```
name=demo
file=<sample A.csv>
file=<b.csv>
```

zsh · exit 0

```
name=demo
file=<sample A.csv>
file=<b.csv>
```

BSH-12-005

Report a missing option value

With a leading colon in optstring, a missing value produces opt=: and OPTARG containing the option letter. The script chooses status 2.

Code

```
set -- -n
OPTIND=1
while getopts ':n:' opt; do
  case "$opt" in
    :) printf 'missing value: -%s\n' "$OPTARG" >&2; exit 2 ;;
  esac
done
```

Actual output

bash · exit 2

```
(no stdout)
```

Standard error

```
missing value: -n
```

zsh · exit 2

```
(no stdout)
```

Standard error

```
missing value: -n
```

BSH-12-006

Report an unknown option

With silent reporting, an unknown option produces opt=? and OPTARG contains its letter. Quote ? in case so it is literal.

Code

```
set -- -x
OPTIND=1
while getopts ':n:' opt; do
  case "$opt" in
    '?') printf 'unknown option: -%s\n' "$OPTARG" >&2; exit 2 ;;
  esac
done
```

Actual output

bash · exit 2

```
(no stdout)
```

Standard error

```
unknown option: -x
```

zsh · exit 2

```
(no stdout)
```

Standard error

```
unknown option: -x
```

Pipeline scope and status details

BSH-13-001

Bash: pipeline assignment disappears

Default Bash runs pipeline components in subshells. The assignment made by read does not reach the parent shell. lastpipe is not enabled.

Code

```
value=before
printf 'after\n' | read -r value
printf '%s\n' "$value"
```

Actual output

bash · exit 0

```
before
```

Zsh difference: final pipeline assignment persists

Native Zsh runs this final builtin in the current shell. The same source therefore changes value.

Code

```
value=before
printf 'after\n' | read -r value
printf '%s\n' "$value"
```

Actual output

zsh · exit 0

```
after
```

BSH-13-003

Use redirection to preserve state

Feed read with process substitution instead of placing it in a pipeline. The builtin then runs in the current shell in both interpreters.

Code

```
value=before
read -r value < <(printf 'after\n')
printf '%s\n' "$value"
```

Actual output

bash · exit 0

```
after
```

zsh · exit 0

```
after
```

BSH-13-004

Bash: inspect every pipeline status

PIPESTATUS is an indexed array of statuses from the latest foreground pipeline. Copy it immediately: the next command replaces it.

Code

```
false | true | false
codes=("${PIPESTATUS[@]}")
printf '%s\n' "${codes[@]}"
```

Actual output

bash · exit 0

```
1
0
1
```

Zsh difference: inspect every pipeline status

The corresponding native Zsh array is pipestatus, in lowercase. Copy it before another command changes it.

Code

```
false | true | false
codes=("${pipestatus[@]}")
printf '%s\n' "${codes[@]}"
```

Actual output

zsh · exit 0

```
1
0
1
```

BSH-13-006

A command substitution trims newlines

Command substitution removes all trailing newlines, not just the final one. Internal newlines remain.

Code

```
value=$(printf 'alpha\n\n')
printf '<%s> length=%s\n' "$value" "${#value}"
```

Actual output

bash · exit 0

```
<alpha> length=5
```

zsh · exit 0

```
<alpha> length=5
```

More patterns and array operations

BSH-14-001

Include hidden files in Bash

dotglob makes * include dotfiles, except . and ..; nullglob removes unmatched patterns. Both settings affect later expansions in this shell.

Code

```
touch visible.txt .hidden.txt
shopt -s dotglob nullglob
files=(*.txt)
printf '%s\n' "${files[@]}"
```

Actual output

bash · exit 0

```
.hidden.txt
visible.txt
```

Zsh difference: Include hidden files in Zsh

Change: the D glob qualifier includes dotfiles for this one pattern; N removes an unmatched pattern.

Code

```
touch visible.txt .hidden.txt
files=(*.txt(DN))
printf '%s\n' "${files[@]}"
```

Actual output

zsh · exit 0

```
.hidden.txt
visible.txt
```

Zsh difference: regular files only

(.) selects regular files; (/) selects directories. N makes either selection empty instead of raising a no-match error.

Code

```
mkdir data
 touch a.txt
printf 'file=%s\n' *(N.)
printf 'dir=%s\n' *(N/)
```

Actual output

zsh · exit 0

```
file=a.txt
dir=data
```

Zsh difference: recurse through directories

**/ searches zero or more directory levels. This example matches both the top-level file and a nested file; N allows no matches.

Code

```
mkdir -p data/raw
touch top.csv data/raw/sample.csv
printf '%s\n' **/*.csv(N)
```

Actual output

zsh · exit 0

```
data/raw/sample.csv
top.csv
```

BSH-14-005

Bash: recursive globbing

globstar enables ** to match directory levels; nullglob handles no matches. Requires Bash 4 or newer.

Code

```
mkdir -p data/raw
touch top.csv data/raw/sample.csv
shopt -s globstar nullglob
printf '%s\n' **/*.csv
```

Actual output

bash · exit 0

```
data/raw/sample.csv
top.csv
```

BSH-14-006

Bash: load lines into an array

mapfile -t reads lines and removes their trailing delimiter. It requires Bash 4 or newer. Empty lines become empty array elements.

Code

```
printf 'alpha\n\nbeta gamma\n' > lines.txt
mapfile -t items < lines.txt
printf 'count=%s\n' "${#items[@]}"
printf '<%s>\n' "${items[@]}"
```

Actual output

bash · exit 0

```
count=3
<alpha>
<>
<beta gamma>
```

Zsh difference: uppercase a scalar

The (U) expansion flag transforms the result to uppercase without assigning back to the variable.

Code

```
label='Sample A'
printf '%s\n' "${(U)label}" "$label"
```

Actual output

zsh · exit 0

```
SAMPLE A
Sample A
```

BSH-14-008

Bash: uppercase a scalar

The ^^ expansion transforms all lowercase characters. Requires Bash 4 or newer; the original variable is unchanged.

Code

```
label='Sample A'
printf '%s\n' "${label^^}" "$label"
```

Actual output

bash · exit 0

```
SAMPLE A
Sample A
```

Debugging and controlled failures

BSH-15-001

Trace expanded commands in Bash

set -x traces commands to stderr after expansion; PS4 sets the trace prefix. set +x stops tracing. Avoid tracing secrets.

Code

```
PS4='+ trace: '
set -x
label=sample
printf '%s\n' "$label"
set +x
```

Actual output

bash · exit 0

```
sample
```

Standard error

```
+ trace: label=sample
+ trace: printf '%s\n' sample
+ trace: set +x
```

Zsh difference: Trace expanded commands in Zsh

Native Zsh also supports set -x and PS4. Formatting can differ; this example is tested independently.

Code

```
PS4='+ trace: '
set -x
label=sample
printf '%s\n' "$label"
set +x
```

Actual output

zsh · exit 0

```
sample
```

Standard error

```
+ trace: label=sample
+ trace: printf '%s\n' sample
+ trace: set +x
```

BSH-15-003

A plain failure under errexit

set -e exits on this unhandled failure. The final printf is never reached. This is only one context of errexit behavior.

Code

```
set -e
printf 'before\n'
false
printf 'unreachable\n'
```

Actual output

bash · exit 1

```
before
```

zsh · exit 1

```
before
```

BSH-15-004

A checked failure under errexit

Change: put false in an if condition. The shell expects a status to test and does not exit.

Code

```
set -e
if false; then
  printf 'unreachable\n'
else
  printf 'handled\n'
fi
printf 'continues\n'
```

Actual output

bash · exit 0

```
handled
continues
```

zsh · exit 0

```
handled
continues
```

BSH-15-005

Capture status before logging

Save $? immediately. Logging with printf would otherwise replace the status you intend to report.

Code

```
false
exit_code=$?
printf 'command failed\n' >&2
printf 'saved=%s now=%s\n' "$exit_code" "$?"
```

Actual output

bash · exit 0

```
saved=1 now=0
```

Standard error

```
command failed
```

zsh · exit 0

```
saved=1 now=0
```

Standard error

```
command failed
```

BSH-15-006

Negation changes the observed status

! inverts success/failure. Inside this branch $? belongs to the negated condition; it is not the original failing status.

Code

```
if ! false; then
  printf 'negated status=%s\n' "$?"
fi
```

Actual output

bash · exit 0

```
negated status=0
```

zsh · exit 0

```
negated status=0
```

Traps, file descriptors and reusable commands

BSH-16-001

Keep a stream on another descriptor

exec 3> opens descriptor 3 in the current shell. >&3 writes to it; exec 3>&- closes it. No new process replaces the shell here.

Code

```
exec 3> log.txt
printf 'log entry\n' >&3
printf 'terminal entry\n'
exec 3>&-
cat log.txt
```

Actual output

bash · exit 0

```
terminal entry
log entry
```

zsh · exit 0

```
terminal entry
log entry
```

BSH-16-002

Preserve failure during cleanup

The EXIT handler records the incoming status, performs cleanup, then returns that same status. The shell retains its explicit exit status of 3.

Code

```
cleanup() {
  local exit_code=$?
  printf 'cleanup after status=%s\n' "$exit_code"
  return "$exit_code"
}
trap 'cleanup' EXIT
exit 3
```

Actual output

bash · exit 3

```
cleanup after status=3
```

zsh · exit 3

```
cleanup after status=3
```

BSH-16-003

Cancel an exit trap

trap - EXIT removes the exit handler. The message registered with the old handler is never printed.

Code

```
trap 'printf "old handler\n"' EXIT
trap - EXIT
printf 'done\n'
```

Actual output

bash · exit 0

```
done
```

zsh · exit 0

```
done
```

BSH-16-004

Store a command as an array

An array preserves the command and each argument. Execute it with "${cmd[@]}". A string plus eval would reparse data as shell syntax.

Code

```
cmd=(printf '<%s>\n' 'sample A' 'sample B')
"${cmd[@]}"
```

Actual output

bash · exit 0

```
<sample A>
<sample B>
```

zsh · exit 0

```
<sample A>
<sample B>
```

BSH-16-005

Source a small function library

The dot builtin evaluates a file in the current shell. Its function definition remains available; source is an equivalent Bash/Zsh spelling.

Code

```
cat > helpers.sh <<'LIB'
greet() { printf 'Hello, %s!\n' "$1"; }
LIB
. ./helpers.sh
greet Ada
```

Actual output

bash · exit 0

```
Hello, Ada!
```

zsh · exit 0

```
Hello, Ada!
```

Zsh difference: Scope options in a Zsh function

emulate -L zsh sets a known option baseline and localizes option changes to the function. Here null_glob does not persist afterward.

Code

```
collect() {
  emulate -L zsh
  setopt null_glob
  local files=(missing*.csv)
  printf 'files=%s\n' "${#files[@]}"
}
collect
[[ -o null_glob ]] || printf 'option restored\n'
```

Actual output

zsh · exit 0

```
files=0
option restored
```

BSH-16-007

Report a failed command with ERR

An ERR trap runs after an unhandled failing command. Save $? as the first action in the handler. The trap reports the status; without errexit, the script continues.

Code

```
trap 'status=$?; printf "ERR: status=%s\n" "$status"' ERR
false
printf 'script continues\n'
```

Actual output

bash · exit 0

```
ERR: status=1
script continues
```

BSH-16-008

A checked failure does not trigger ERR

Change: test false in an if condition. The else branch handles its status, so the ERR trap does not run. A trap cannot replace explicit checks in these contexts.

Code

```
trap 'status=$?; printf "ERR: status=%s\n" "$status"' ERR
if false; then
  printf 'unexpected success\n'
else
  printf 'failure handled\n'
fi
```

Actual output

bash · exit 0

```
failure handled
```

BSH-16-009

Keep ERR traps outside a function

set +E disables ERR-trap inheritance. The failure inside the function is followed by a successful printf, so the function itself returns zero and no outer ERR trap fires.

Code

```
set +E
trap 'status=$?; printf "ERR: status=%s\n" "$status"' ERR
run_step() {
  false
  printf 'function continues\n'
}
run_step
printf 'caller continues\n'
```

Actual output

bash · exit 0

```
function continues
caller continues
```

BSH-16-010

Inherit ERR traps inside a function

Change only +E to -E. Error tracing inherits the ERR trap inside the function, so its unchecked false is reported. This option enables inheritance, not automatic exit.

Code

```
set -E
trap 'status=$?; printf "ERR: status=%s\n" "$status"' ERR
run_step() {
  false
  printf 'function continues\n'
}
run_step
printf 'caller continues\n'
```

Actual output

bash · exit 0

```
ERR: status=1
function continues
caller continues
```

BSH-16-011

Combine an ERR report with errexit

set -e exits after this unchecked false. The ERR handler reports status 1 to stderr before exit; the final printf is never reached. The example intentionally exits 1. Checked conditions still follow the exception illustrated in BSH-16-008.

Code

```
set -e
trap 'status=$?; printf "ERR: status=%s\n" "$status" >&2' ERR
printf 'before failure\n'
false
printf 'unreachable\n'
```

Actual output

bash · exit 1

```
before failure
```

Standard error

```
ERR: status=1
```

Strings, formats and regular expressions

BSH-17-001

Take a substring

${text:offset:length} extracts length characters starting at zero-based offset. It does not change text.

Zsh difference: Scalar substring offsets are also zero-based in Zsh; this differs from native Zsh array indexing.

Code

```
text='sample-2026.csv'
printf '%s\n' "${text:7:4}"
```

Actual output

bash · exit 0

```
2026
```

BSH-17-002

Count from the end

A negative offset counts from the end. Keep a space before -4 to distinguish this from the :- default-value operator.

Code

```
text='sample-2026.csv'
printf '%s\n' "${text: -4}"
```

Actual output

bash · exit 0

```
.csv
```

BSH-17-003

Format directly into a variable

Bash printf -v assigns its formatted output without a command substitution. %03d pads to at least three digits.

Code

```
printf -v sample 'sample-%03d' 7
printf '%s\n' "$sample"
```

Actual output

bash · exit 0

```
sample-007
```

BSH-17-004

Show an argument as reusable shell syntax

printf %q produces a representation that can be read as one Bash argument. This is shell quoting, not JSON or URL encoding.

Zsh difference: Zsh also offers quoting facilities, but printed quoting styles can differ. Do not use a %q representation as a cross-language serialization format.

Code

```
file='sample A.csv'
printf '%q\n' "$file"
```

Actual output

bash · exit 0

```
sample\ A.csv
```

BSH-17-005

Match an extended regular expression

=~ uses an extended regular expression. ^ and $ anchor the whole string; + requires one or more digits. Leave the regex variable unquoted on the right.

Code

```
value='123'
pattern='^[0-9]+$'
if [[ "$value" =~ $pattern ]]; then
  printf 'integer text\n'
fi
```

Actual output

bash · exit 0

```
integer text
```

BSH-17-006

Capture groups from a regex

BASH_REMATCH[0] is the entire match; [1] and [2] contain parenthesized groups. Read captures only after a successful match.

Zsh difference: Native Zsh normally uses MATCH for the full match and match[1], match[2], ... for captures. BASH_REMATCH is not its default capture interface.

Code

```
file='sample-042.csv'
pattern='^([a-z]+)-([0-9]+)\.csv$'
if [[ "$file" =~ $pattern ]]; then
  printf 'label=%s number=%s\n' \
    "${BASH_REMATCH[1]}" "${BASH_REMATCH[2]}"
fi
```

Actual output

bash · exit 0

```
label=sample number=042
```

BSH-17-007

Quote a regex to match it literally

Change: quote the expanded pattern on the right of =~. Its metacharacters lose their regex meaning.

Code

```
value='123'
pattern='^[0-9]+$'
if [[ "$value" =~ "$pattern" ]]; then
  printf 'literal match\n'
else
  printf 'no literal match\n'
fi
```

Actual output

bash · exit 0

```
no literal match
```

Arithmetic and loop control

BSH-18-001

Arithmetic commands return a status

((expression)) succeeds when the value is nonzero. Zero produces status 1. This can interact unexpectedly with set -e.

Code

```
(( 2 + 3 ))
printf 'nonzero status=%s\n' "$?"
(( 2 - 2 ))
printf 'zero status=%s\n' "$?"
```

Actual output

bash · exit 0

```
nonzero status=0
zero status=1
```

BSH-18-002

Postincrement versus preincrement

n++ evaluates to the old value, whereas ++n evaluates to the new value. Both increment n, but their exit statuses can differ.

Code

```
n=0
((n++))
printf 'post: n=%s status=%s\n' "$n" "$?"
n=0
((++n))
printf 'pre: n=%s status=%s\n' "$n" "$?"
```

Actual output

bash · exit 0

```
post: n=1 status=1
pre: n=1 status=0
```

BSH-18-003

Read a zero-padded decimal integer

10# forces base ten. Without an explicit base, a leading zero denotes octal in Bash arithmetic. Validate input before arithmetic evaluation.

Code

```
value=008
printf '%s\n' "$((10#$value + 1))"
```

Actual output

bash · exit 0

```
9
```

BSH-18-004

Use an arithmetic for loop

The three clauses initialize, test and advance the counter. This loop prints 0, 1 and 2.

Code

```
for ((i=0; i<3; i++)); do
  printf '%s\n' "$i"
done
```

Actual output

bash · exit 0

```
0
1
2
```

BSH-18-005

Stop at the first match

break leaves the nearest enclosing loop. Here the final iteration is never reached.

Code

```
for value in 3 7 9; do
  if ((value > 5)); then
    printf 'first=%s\n' "$value"
    break
  fi
done
```

Actual output

bash · exit 0

```
first=7
```

BSH-18-006

Compute a decimal result with a utility

Bash arithmetic is integer arithmetic. Use a numeric tool for decimal work; awk is used here with C-locale output.

Zsh difference: Zsh supports floating-point arithmetic natively, for example $((7.0 / 2)). Bash does not accept that expression.

Code

```
awk 'BEGIN {printf "%.2f\n", 7 / 2}'
```

Actual output

bash · exit 0

```
3.50
```

Arrays, indirection and references

BSH-19-001

Array indices need not be contiguous

Assigning index 5 does not create indices 0-4. ${#items[@]} counts assigned elements, not the largest index plus one.

Zsh difference: Native Zsh arrays use different indexing and gap semantics. Do not translate sparse Bash arrays by simply adding one to each index.

Code

```
items=()
items[2]=alpha
items[5]=beta
printf 'count=%s\n' "${#items[@]}"
printf 'index=%s\n' "${!items[@]}"
```

Actual output

bash · exit 0

```
count=2
index=2
index=5
```

BSH-19-002

Unset one array element

Quote the unset argument so the brackets are not treated as filename syntax. Remaining Bash array indices are not renumbered.

Zsh difference: Native Zsh array element deletion and indexing differ; the sparse-array behavior shown here is Bash-specific.

Code

```
items=(alpha beta gamma)
unset 'items[1]'
printf 'index=%s\n' "${!items[@]}"
printf 'value=%s\n' "${items[@]}"
```

Actual output

bash · exit 0

```
index=0
index=2
value=alpha
value=gamma
```

BSH-19-003

Slice several array elements

${items[@]:offset:length} selects length elements. Quotes keep each selected element separate.

Zsh difference: Zsh native array subscript ranges are inclusive and normally one-based, for example "${items[2,3]}". Do not confuse them with Bash offset/length notation.

Code

```
items=(alpha 'beta gamma' delta epsilon)
printf '<%s>\n' "${items[@]:1:2}"
```

Actual output

bash · exit 0

```
<beta gamma>
<delta>
```

BSH-19-004

Look up a variable by name

${!name} reads the variable whose name is stored in name. Use known variable names; this is not a reason to use eval.

Zsh difference: Zsh uses the (P) parameter expansion flag for this form of indirection, for example "${(P)name}".

Code

```
label=sample
name=label
printf '%s\n' "${!name}"
```

Actual output

bash · exit 0

```
sample
```

BSH-19-005

Update through a nameref

declare -n creates a reference to another variable. Writing through ref updates label. Requires Bash 4.3 or newer.

Zsh difference: Do not assume Bash declare -n is available in Zsh; this syntax is not supported by the Zsh 5.9 comparison environment.

Code

```
label=before
declare -n ref=label
ref=after
printf '%s\n' "$label"
```

Actual output

bash · exit 0

```
after
```

BSH-19-006

Return an array through a function argument

A local nameref lets a function populate the caller-supplied array without flattening it into text. Requires Bash 4.3+. Use a reference name that cannot collide with the target.

Zsh difference: This uses Bash namerefs; it is not a portable Zsh function interface.

Code

```
make_items() {
  local -n output_ref="$1"
  output_ref=(alpha 'beta gamma')
}
items=()
make_items items
printf '<%s>\n' "${items[@]}"
```

Actual output

bash · exit 0

```
<alpha>
<beta gamma>
```

BSH-19-007

List variable names by prefix

${!tour_demo_@} expands to variable names beginning with tour_demo_, rather than their values. Quoted @ expansion keeps each name separate. sort makes the display order explicit.

Code

```
tour_demo_alpha=12
tour_demo_beta=7
printf '%s\n' "${!tour_demo_@}" | sort
```

Actual output

bash · exit 0

```
tour_demo_alpha
tour_demo_beta
```

BSH-19-008

Read values through discovered names

Change: iterate over the discovered names and use ${!variable} to read each value. The names come from local variables with the chosen prefix; no eval or command-string construction is needed. sort fixes the presentation order.

Code

```
tour_demo_alpha=12
tour_demo_beta=7
for variable in "${!tour_demo_@}"; do
  printf '%s=%s\n' "$variable" "${!variable}"
done | sort
```

Actual output

bash · exit 0

```
tour_demo_alpha=12
tour_demo_beta=7
```

Options and controlled file output

BSH-20-001

Protect an existing output file

set -C enables noclobber: > refuses to overwrite an existing regular file. Catch the status explicitly. Error text is suppressed here so the intentional outcome is clear.

Code

```
printf 'original\n' > report.txt
set -C
if { printf 'replacement\n' > report.txt; } 2>/dev/null; then
  printf 'written\n'
else
  printf 'overwrite refused\n'
fi
cat report.txt
```

Actual output

bash · exit 0

```
overwrite refused
original
```

BSH-20-002

Override noclobber deliberately

Change: >| permits this overwrite even when noclobber is enabled. It is a deliberate exception for this redirection.

Code

```
printf 'original\n' > report.txt
set -C
printf 'replacement\n' >| report.txt
cat report.txt
```

Actual output

bash · exit 0

```
replacement
```

BSH-20-003

Scope a Bash option with a subshell

Set nullglob inside (...) to keep the option change local. The parent retains its previous value.

Zsh difference: Zsh uses setopt rather than shopt; its emulate -L zsh idiom localizes options inside a function.

Code

```
shopt -u nullglob
(
  shopt -s nullglob
  files=(missing*.csv)
  printf 'inside=%s\n' "${#files[@]}"
)
shopt -q nullglob || printf 'parent unchanged\n'
```

Actual output

bash · exit 0

```
inside=0
parent unchanged
```

BSH-20-004

Match case-insensitively

nocasematch affects [[...]] and case matching. It is distinct from nocaseglob, which changes filename expansion.

Zsh difference: The equivalent native Zsh option is NO_CASE_MATCH, set with setopt nocasematch.

Code

```
shopt -s nocasematch
file='SAMPLE.CSV'
if [[ "$file" == *.csv ]]; then printf 'matched\n'; fi
```

Actual output

bash · exit 0

```
matched
```

BSH-20-005

Keep the last pipeline builtin in Bash

lastpipe lets the last foreground pipeline command run in the current shell when job control is disabled. Requires Bash 4.2+.

Zsh difference: Native Zsh already runs this final builtin in the current shell. It does not require Bash lastpipe.

Code

```
set +m
shopt -s lastpipe
value=before
printf 'after\n' | read -r value
printf '%s\n' "$value"
```

Actual output

bash · exit 0

```
after
```

BSH-20-006

Disable filename expansion

set -f disables globbing. The literal asterisk reaches printf even though matching files exist. set +f enables globbing again.

Code

```
touch a.txt b.txt
set -f
printf '%s\n' *.txt
set +f
printf '%s\n' *.txt
```

Actual output

bash · exit 0

```
*.txt
a.txt
b.txt
```

Reusable script patterns

BSH-21-001

Check a dependency before using it

command -v tests whether a command is available. Redirect both streams when only its status matters.

Code

```
if command -v awk >/dev/null 2>&1; then
  printf 'awk available\n'
else
  printf 'awk missing\n' >&2
  exit 1
fi
```

Actual output

bash · exit 0

```
awk available
```

BSH-21-002

Validate an argument count

Use $# before accessing required arguments. A function can return a usage-error status without exiting its caller.

Code

```
require_one() {
  if (( $# != 1 )); then
    printf 'expected one argument\n' >&2
    return 2
  fi
  printf 'input=%s\n' "$1"
}
require_one 'sample A'
require_one
printf 'validation status=%s\n' "$?"
```

Actual output

bash · exit 0

```
input=sample A
validation status=2
```

Standard error

```
expected one argument
```

BSH-21-003

Separate data from diagnostic messages

stdout remains suitable for further processing; diagnostics go to stderr. Capture each stream independently.

Code

```
produce() {
  printf 'processing sample\n' >&2
  printf '7\n'
}
produce > values.txt 2> progress.log
printf 'DATA: '; cat values.txt
printf 'LOG: '; cat progress.log
```

Actual output

bash · exit 0

```
DATA: 7
LOG: processing sample
```

BSH-21-004

A dry-run wrapper preserves arguments

The wrapper shows the argument boundaries using %q, then optionally executes the original array of arguments. It does not reparse the printed representation.

Code

```
run() {
  if [[ "$dry_run" == yes ]]; then
    printf 'would run:'
    printf ' %q' "$@"
    printf '\n'
  else
    "$@"
  fi
}
dry_run=yes
run printf '%s\n' 'sample A'
```

Actual output

bash · exit 0

```
would run: printf %s\\n sample\ A
```

BSH-21-005

Test a function result explicitly

A small assertion compares the computed value with an independently stated expected result. On failure the script would exit nonzero.

Code

```
stem() { printf '%s\n' "${1%.csv}"; }
actual=$(stem sample.csv)
if [[ "$actual" != sample ]]; then
  printf 'FAIL: stem\n' >&2
  exit 1
fi
printf 'PASS: stem\n'
```

Actual output

bash · exit 0

```
PASS: stem
```

Example and command index

- BSH-01-001 · Print a line
- BSH-01-002 · One argument, one line
- BSH-01-003 · Keep words together
- BSH-01-004 · Align columns
- BSH-01-005 · Choose decimal precision
- BSH-01-006 · Comments and continuation
- BSH-02-001 · Assign and expand
- BSH-02-002 · Single versus double quotes
- BSH-02-003 · Bash splits an unquoted scalar
- BSH-02-004 · Zsh difference: keeps an unquoted scalar
- BSH-02-005 · Zsh difference: Explicit Zsh splitting
- BSH-02-006 · Protect a variable boundary
- BSH-03-001 · A fallback value
- BSH-03-002 · Unset versus empty
- BSH-03-003 · Assign the fallback
- BSH-03-004 · Remove a suffix
- BSH-03-005 · Shortest versus longest match
- BSH-03-006 · Remove a path prefix
- BSH-03-007 · Replace matching text
- BSH-03-008 · Integer arithmetic
- BSH-04-001 · Create, copy and read
- BSH-04-002 · Select a filename suffix
- BSH-04-003 · A quoted pattern stays literal
- BSH-04-004 · Exactly one character
- BSH-04-005 · Brace expansion does not find files
- BSH-04-006 · Bash unmatched glob
- BSH-04-007 · Zsh difference: unmatched glob
- BSH-04-008 · Bash: collect zero matches
- BSH-04-009 · Zsh difference: collect zero matches
- BSH-05-001 · Write a file
- BSH-05-002 · Append instead
- BSH-05-003 · Read standard input
- BSH-05-004 · Connect two commands
- BSH-05-005 · Sort and count repeats
- BSH-05-006 · Separate stdout and stderr
- BSH-05-007 · Capture both streams
- BSH-05-008 · Redirection order matters
- BSH-05-009 · A literal here-document
- BSH-05-010 · An expanding here-document
- BSH-05-011 · Save and continue a stream
- BSH-05-012 · Feed a variable through a here-string
- BSH-05-013 · An empty here-string still supplies a line
- BSH-06-001 · Read the exit status
- BSH-06-002 · Run only after success
- BSH-06-003 · Recover after failure
- BSH-06-004 · Branch on a number
- BSH-06-005 · Match a string pattern
- BSH-06-006 · Several alternatives with case
- BSH-06-007 · Default pipeline status
- BSH-06-008 · Enable pipefail
- BSH-07-001 · Loop over explicit arguments
- BSH-07-002 · Read complete lines
- BSH-07-003 · Repeat while a condition holds
- BSH-07-004 · Preserve positional arguments

- BSH-07-005 · Join positional arguments
- BSH-07-006 · A function with an argument
- BSH-07-007 · Capture a command result
- BSH-07-008 · Return a status from a function
- BSH-08-001 · Bash starts at index zero
- BSH-08-002 · Zsh difference: starts at index one
- BSH-08-003 · Preserve array elements
- BSH-08-004 · Append an array element
- BSH-08-005 · Bash associative array
- BSH-08-006 · Zsh difference: associative array
- BSH-08-007 · Zsh difference: split and join flags
- BSH-08-008 · Zsh difference: path modifiers
- BSH-09-001 · A subshell isolates assignments
- BSH-09-002 · A group shares assignments
- BSH-09-003 · Export to a child process
- BSH-09-004 · Wait for a background job
- BSH-09-005 · Process substitution
- BSH-09-006 · Unset variables become errors
- BSH-09-007 · A fallback works with nounset
- BSH-09-008 · Cleanup on shell exit
- BSH-10-001 · Stage a tiny dataset
- BSH-10-002 · Validate before processing
- BSH-10-003 · Compute and write a report
- BSH-11-001 · Keep a space inside a filename
- BSH-11-002 · A leading dash is still a filename
- BSH-11-003 · Read NUL-separated records
- BSH-11-004 · Process the last unterminated line
- BSH-11-005 · Read two fields
- BSH-11-006 · Skip a file deliberately
- BSH-12-001 · Consume one argument
- BSH-12-002 · Parse a short flag
- BSH-12-003 · An option with a value
- BSH-12-004 · Keep the remaining filenames
- BSH-12-005 · Report a missing option value
- BSH-12-006 · Report an unknown option
- BSH-13-001 · Bash: pipeline assignment disappears
- BSH-13-002 · Zsh difference: final pipeline assignment persists
- BSH-13-003 · Use redirection to preserve state
- BSH-13-004 · Bash: inspect every pipeline status
- BSH-13-005 · Zsh difference: inspect every pipeline status
- BSH-13-006 · A command substitution trims newlines
- BSH-14-001 · Include hidden files in Bash
- BSH-14-002 · Zsh difference: Include hidden files in Zsh
- BSH-14-003 · Zsh difference: regular files only
- BSH-14-004 · Zsh difference: recurse through directories
- BSH-14-005 · Bash: recursive globbing
- BSH-14-006 · Bash: load lines into an array
- BSH-14-007 · Zsh difference: uppercase a scalar
- BSH-14-008 · Bash: uppercase a scalar
- BSH-15-001 · Trace expanded commands in Bash
- BSH-15-002 · Zsh difference: Trace expanded commands in Zsh
- BSH-15-003 · A plain failure under errexit
- BSH-15-004 · A checked failure under errexit
- BSH-15-005 · Capture status before logging
- BSH-15-006 · Negation changes the observed status
- BSH-16-001 · Keep a stream on another descriptor
- BSH-16-002 · Preserve failure during cleanup
- BSH-16-003 · Cancel an exit trap
- BSH-16-004 · Store a command as an array
- BSH-16-005 · Source a small function library
- BSH-16-006 · Zsh difference: Scope options in a Zsh function
- BSH-16-007 · Report a failed command with ERR
- BSH-16-008 · A checked failure does not trigger ERR
- BSH-16-009 · Keep ERR traps outside a function
- BSH-16-010 · Inherit ERR traps inside a function
- BSH-16-011 · Combine an ERR report with errexit
- BSH-17-001 · Take a substring
- BSH-17-002 · Count from the end

- BSH-17-003 · Format directly into a variable
- BSH-17-004 · Show an argument as reusable shell syntax
- BSH-17-005 · Match an extended regular expression
- BSH-17-006 · Capture groups from a regex
- BSH-17-007 · Quote a regex to match it literally
- BSH-18-001 · Arithmetic commands return a status
- BSH-18-002 · Postincrement versus preincrement
- BSH-18-003 · Read a zero-padded decimal integer
- BSH-18-004 · Use an arithmetic for loop
- BSH-18-005 · Stop at the first match
- BSH-18-006 · Compute a decimal result with a utility
- BSH-19-001 · Array indices need not be contiguous
- BSH-19-002 · Unset one array element
- BSH-19-003 · Slice several array elements
- BSH-19-004 · Look up a variable by name
- BSH-19-005 · Update through a nameref
- BSH-19-006 · Return an array through a function argument
- BSH-19-007 · List variable names by prefix
- BSH-19-008 · Read values through discovered names
- BSH-20-001 · Protect an existing output file
- BSH-20-002 · Override noclobber deliberately
- BSH-20-003 · Scope a Bash option with a subshell
- BSH-20-004 · Match case-insensitively
- BSH-20-005 · Keep the last pipeline builtin in Bash
- BSH-20-006 · Disable filename expansion
- BSH-21-001 · Check a dependency before using it
- BSH-21-002 · Validate an argument count
- BSH-21-003 · Separate data from diagnostic messages
- BSH-21-004 · A dry-run wrapper preserves arguments
- BSH-21-005 · Test a function result explicitly

Content: CC0 1.0. Structure and publishing framework: CC BY 4.0 by Lukas C. Bossert. Repository source · Publishing framework source · License scope and third-party notices.