Contents

shell-cmd — Complete Guide

What Is shell-cmd?

`shell-cmd` is a command-line utility written in C++20 that **recursively walks a directory tree**, finds files matching a **regex pattern**, and **executes a shell command for each match**. Think of it as a more ergonomic alternative to chaining `find` with `-exec` — you write a command template with numbered placeholders (`%0`, `%1`, `%2`, ...) and `shell-cmd` fills them in and runs the command for every matched file.

How the Program Works (Internals)

High-Level Flow

```
1. Parse command-line options and positional arguments
2. Recursively walk the target directory
3. If --list-all mode (-l):
   a. Collect all matching file paths into a list
   b. Run the command template once with %0 = all matched paths joined by spaces
4. Otherwise (default per-file mode):
   a. For each file whose full path matches the regex:
      i.  Substitute placeholders in the command template
      ii. Fork a child process and execute the command via the configured shell
```

Argument Parsing

`shell-cmd` uses a custom header-only argument parser (`argz.hpp`). It separates **options** (short like `-n` or long like `--dry-run`) from **positional arguments**. All options support both forms. The positional arguments are, in order:

Position	Meaning
1st	**Path** — the root directory to search
2nd	**Command template** — shell command with `%` placeholders
3rd	**Regex** — ECMAScript regex matched against the full file path (optional when `--expr` is used)
4th+	**Extra arguments** — substituted into `%2`, `%3`, etc.

If fewer than three positional arguments are given (and `--expr` is not set), the program prints an error and the help text. When `--expr` is used, the regex argument is not required — only the path and command template are needed.

Directory Traversal

The function `add_directory()` walks the directory tree using `std::filesystem::directory_iterator`. Key behaviors:

- **Hidden files/directories** (names starting with `.`) are **skipped by default**. Pass `-a` to include them.
- **Depth limiting** — if `-d N` is specified, recursion stops after `N` levels (0 means only the given directory, no subdirectories).
- **Permission errors** are handled gracefully; directories that can't be opened produce an error message and exit.
- **Symlinks** are not explicitly followed — standard `directory_iterator` behavior applies.

For every regular file whose full path matches the regex (or satisfies the `--expr` expression), the program calls `proc_cmd()`.

In **list-all mode** (`-l` / `--list-all`), a separate function `fill_list()` walks the tree and collects all matching paths into a vector. After the walk completes, the paths are joined with spaces and `proc_cmd()` is called once with `%0` expanded to the full list.

Placeholder Substitution

Inside `proc_cmd()`, the command template string is scanned and placeholders are replaced:

Placeholder	Replaced With
`%0`	The **filename only** (no directory path), extracted via `std::filesystem::path::filename()`
`%1`	The **full path** to the matched file
`%b`	The **basename without extension**, extracted via `std::filesystem::path::stem()`
`%e`	The **file extension** (including the dot), extracted via `std::filesystem::path::extension()`
`%2`, `%3`, ...	The **extra arguments** passed after the regex on the command line

If the full path for `%1` contains spaces, it is automatically wrapped in double quotes to prevent word-splitting by the shell.

Command Execution

Commands are executed through a custom `System()` function. Rather than calling the standard library's `system()`, this implementation:

1. **Forks** a child process.
2. **Blocks** `SIGCHLD` and **ignores** `SIGINT`/`SIGQUIT` in the parent so the parent isn't accidentally killed by Ctrl+C.
3. Runs the command via `execl("/bin/bash", "bash", "-c", command, ...)` in the child (or the shell specified by `--shell`).
4. **Waits** for the child to finish, then restores signal masks.

This gives reliable process management and prevents interrupted batch operations from leaving the parent in a bad state.

When **parallel mode** (`-j N`) is active, `proc_cmd()` forks child processes directly and maintains a pool of up to N concurrent workers, bypassing the `System()` function. The `wait_for_slot()` and `wait_all()` helpers manage the process pool.

Building from Source

Prerequisites

- A C++20-capable compiler (GCC 13+ or Clang 16+)
- CMake 3.10+ (for the CMake build)

CMake Build (recommended)

```
cd shell-cmd
mkdir -p build && cd build
cmake ..
make
```

The compiled binary is at `build/shell-cmd`.

System-Wide Install

```
cd build
sudo make install
```

This copies the binary to `/usr/local/bin` (or the system default `bin` directory).

Alternative: Plain Makefile

```
make -f Makefile.cmd
sudo make -f Makefile.cmd install
```

Usage Synopsis

`shell-cmd [options] <path> "<command %1 [%2 %3..]>" <regex> [extra_args..]`

Options

Short	Long	Description
`-z`	`--regex-match`	**Regex match** — use `regex_match` (full path must match) instead of `regex_search`
`-b`	`--glob`	**Glob mode** — treat pattern as a glob (`*`, `?`) instead of regex
`-n`	`--dry-run`	**Dry-run** — print each command but don't execute it
`-v`	`--verbose`	**Verbose** — print each command before executing it
`-a`	`--all`	**All files** — include hidden files and directories
`-l`	`--list-all`	**List all** — collect all matches and run command once with `%0` = all matched paths
`-d N`	`--depth N`	**Max depth** — limit recursion (0 = current directory only)
`-s SIZE`	`--size SIZE`	**Size filter** — `+10M` (>10 MB), `-1K` (<1 KB), `4096` (exact bytes). Suffixes: K, M, G
`-m DAYS`	`--mtime DAYS`	**Modification time** — `+7` (older than 7 days), `-1` (within last day), `3` (exactly 3 days)
`-p MODE`	`--perm MODE`	**Permissions** — octal mode, e.g. `755`

Short	Long	Description
`-u USER`	`--user USER`	**Owner** — filter by username
`-g GROUP`	`--group GROUP`	**Group** — filter by group name
`-t TYPE`	`--type TYPE`	**Type** — `f` (file), `d` (directory), `l` (symlink)
`-x REGEX`	`--exclude REGEX`	**Exclude** — skip files/directories matching the regex
`-i`	`--glob-exclude`	**Glob exclude** — treat the exclude pattern as a glob instead of regex
`-f EXPR`	`--expr EXPR`	**Expression filter** — compose `glob()`, `regex()`, `regex_match()` with `and`/`or`/`not` (replaces the regex positional argument)
`-e`	`--stop-on-error`	**Stop on error** — halt on first command failure
`-c`	`--confirm`	**Confirm** — prompt yes/no before each command
`-j N`	`--jobs N`	**Parallel** — run N commands concurrently (default: 1)
`-w SHELL`	`--shell SHELL`	**Shell** — shell to use for execution (default: `/bin/bash`)
`-h`	`--help`	**Help** — show usage information

Concrete Real-Life Examples

1. Count Lines of Code in a Project

You want to see line counts for every `.py` file in your project:

```
shell-cmd . "wc -l %1" ".*\.py$"
```

What happens: Every `.py` file found recursively under `.` is passed to `wc -l`. Output looks like:

```
  42 ./src/main.py
 118 ./src/utils.py
  27 ./tests/test_main.py
```

2. Preview Before You Act (Dry-Run)

You want to auto-format all C/C++ source files, but first check what would run:

```
shell-cmd -n ./src "clang-format -i %1" ".*\.(c|cpp|h|hpp)$"
```

Output (nothing is executed):

```
clang-format -i ./src/main.cpp
clang-format -i ./src/parser.hpp
clang-format -i ./src/utils.c
```

When satisfied, remove `-n` to actually format the files:

```
shell-cmd ./src "clang-format -i %1" ".*\.(c|cpp|h|hpp)$"
```

3. Batch Resize Photos

Resize every JPEG in your photo library to 1920×1080 using ImageMagick:

```
shell-cmd ~/Photos "convert %1 -resize 1920x1080 /tmp/resized/%0" ".*\.jpe?g$"
```

- `%1` → `/home/you/Photos/vacation/sunset.jpg` (the source)

- `%0` → `sunset.jpg` (used to name the output file)

The regex `jpe?g` matches both `.jpg` and `.jpeg`.

4. Back Up Log Files to Another Directory

Copy all `.log` files to a backup folder, preserving filenames:

```
shell-cmd /var/log "cp %1 %2/%0" ".*\.log$" /mnt/backup/logs
```

- `%1` → full path to each log file
- `%2` → `/mnt/backup/logs` (the extra argument)
- `%0` → the filename only

5. Search for TODO Comments Across a Codebase

```
shell-cmd . "grep -Hn 'TODO' %1" ".*\.(js|ts|py|cpp)$"
```

This runs `grep` with line numbers on every source file. Example output:

```
./src/api.ts:45:    // TODO: add rate limiting
./src/db.py:112:    # TODO: handle connection timeout
```

6. Convert All Markdown Files to PDF

Using Pandoc to generate PDFs from your documentation:

```
shell-cmd ~/docs "pandoc %1 -o /tmp/pdfs/%0.pdf" ".*\.md$"
```

Each Markdown file becomes a PDF in `/tmp/pdfs/`.

7. Transcode WAV Audio to MP3

```
shell-cmd ~/recordings "ffmpeg -i %1 -b:a 192k /tmp/mp3/%0.mp3" ".*\.wav$"
```

8. Validate Shell Scripts Without Running Them

Check syntax of all `.sh` files with verbose output:

```
shell-cmd -v . "bash -n %1" ".*\.sh$"
```

`-v` prints each command as it runs, so you see:

```
bash -n ./deploy.sh
bash -n ./setup.sh
bash -n ./scripts/cleanup.sh
```

If any script has a syntax error, `bash -n` will report it.

9. Extract All tar.gz Archives

```
shell-cmd ~/Downloads "tar xzf %1 -C /tmp/extracted" ".*\.tar\.gz$"
```

10. Strip EXIF Metadata Before Sharing Photos

```
shell-cmd ./photos "exiftool -all= %1" ".*\.(jpg|png)$"
```

11. Work Only in the Current Directory (No Recursion)

Limit depth to 0 so only files directly in the given path are processed:

```
shell-cmd -d 0 . "cat %1" ".*\.txt$"
```

12. Include Hidden Config Files

Normally `shell-cmd` skips dotfiles. Use `-a` to include them:

```
shell-cmd -a ~ "cat %1" ".*\.bashrc|.*\.zshrc"
```

13. Combine Multiple Options

Preview commands, include hidden files, limit depth to 2 levels:

```
shell-cmd -n -a -d 2 ~ "wc -l %1" ".*rc$"
```

Output (dry-run, nothing executed):

```
wc -l /home/you/.bashrc
wc -l /home/you/.config/i3/config  (skipped - depth > 2 or no match)
wc -l /home/you/.zshrc
```

14. Compile Every C File in a Project

```
shell-cmd ./src "gcc -c %1 -o /tmp/%0.o" ".*\.c$"
```

- `%1` gives `gcc` the full source path
- `%0.o` names the output object file after the source filename

15. Sign All RPM Packages in a Repository

```
shell-cmd ./packages "rpm --addsign %1" ".*\.rpm$"
```

16. Using Multiple Extra Arguments

You can pass multiple extra arguments after the regex. Each maps to `%2`, `%3`, etc.:

```
shell-cmd . "cp %1 %2/%0 && echo 'copied to %3'" ".*\.conf$" /backup user@host
```

- `%2` → `/backup`
- `%3` → `user@host`

The program validates that every extra argument has a corresponding placeholder in the command template. If you pass an extra argument but the command doesn't reference its placeholder, `shell-cmd` exits with an error.

Metadata Filter Examples

17. Find Large Files

List all files over 10 MB:

```
shell-cmd . "ls -lh %1" ".*" --size +10M
```

Or equivalently with short flags:

```
shell-cmd . "ls -lh %1" ".*" -s +10M
```

18. List All Matches in One Command (list-all)

Use `-l`/`--list-all` when you want a single command invocation with all matches joined into one string and substituted by `%0`:

```
shell-cmd -l . "echo Found files: %0" ".*\.log$"
```

In this case, `shell-cmd` first collects all matched files and then executes only one command, instead of running the command once per file.

19. Delete Old Temp Files (Dry-Run)

Preview deleting `.tmp` files older than 30 days:

```
shell-cmd --dry-run /tmp "rm %1" ".*\.tmp$" --mtime +30
```

20. Find Executable Files

Find files with permission `755`:

```
shell-cmd . "echo %1" ".*" --perm 755 --type f
```

21. List Files Owned by a User

```
shell-cmd /etc "echo %1" ".*\.conf$" --user root
```

22. Find Files by Group

```
shell-cmd /var/www "echo %1" ".*" --group www-data
```

23. List Only Directories

```
shell-cmd . "echo %1" ".*src.*" --type d
```

24. Find Symlinks

```
shell-cmd /usr/local "ls -la %1" ".*" --type l
```

25. Combine Multiple Filters

Find large `.log` files modified in the last 7 days, owned by `syslog`:

```
shell-cmd /var/log "wc -l %1" ".*\.log$" -s +1M -m -7 -u syslog
```

26. Long-Form Options Only

All flags work with `--long` form for readability in scripts:

```
shell-cmd --verbose --size +5K --type f --depth 2 ./src "wc -l %1" ".*\.(cpp|hpp)$"
```

New in v1.2

27. Exclude Patterns

Skip `node_modules` and `.git` directories when counting TypeScript lines:

```
shell-cmd -x "node_modules|\.git" . "wc -l %1" ".*\.ts$"
```

28. Glob Mode

Use `--glob` / `-b` to write familiar wildcard patterns instead of regex. `*` matches anything, `?` matches a single character, and special regex characters (`.`, `+`, `(`, etc.) are auto-escaped:

```
shell-cmd --glob . "echo %1" "*.cpp"
```

Combine with `--regex-match` to match the full path using glob syntax:

```
shell-cmd --glob --regex-match . "echo %1" "*cmake"
```

Glob also applies to `--exclude` when combined with `--glob-exclude` / `-i`:

```
shell-cmd --glob -x "*.o" --glob-exclude . "echo %1" "*.c"
```

Without `--glob-exclude`, the `-x` pattern is always treated as a regex:

```
shell-cmd --glob -x "build|CMakeFiles" . "echo %1" "*.cpp"
```

28. Basename & Extension Placeholders

Convert WAV audio files to MP3, using `%b` to name the output file without the original extension:

```
shell-cmd ~/music "ffmpeg -i %1 /tmp/mp3/%b.mp3" ".*\.wav$"
```

Extract extensions to organize files by type:

```
shell-cmd -n . "mkdir -p /tmp/by-ext/%e && cp %1 /tmp/by-ext/%e/%0" ".*"
```

29. Parallel Execution

Resize images using 4 parallel jobs:

```
shell-cmd -j 4 ./images "convert %1 -resize 800x600 /tmp/thumbs/%0" ".*\.jpg$"
```

30. Confirm Mode

Interactively confirm before each destructive action:

```
shell-cmd -c /tmp "rm %1" ".*\.bak$"
```

Output:

```
Execute: rm /tmp/old.bak ? [y/N]
```

31. Stop on Error

Compile all C files and stop at the first failure:

```
shell-cmd -e ./src "gcc -c %1 -o /tmp/%b.o" ".*\.c$"
```

If any `gcc` invocation returns non-zero, processing halts immediately.

32. Summary Statistics

A summary line is automatically printed to stderr after execution when verbose, dry-run, or any command has failed:

```
shell-cmd -v . "wc -l %1" ".*\.py$"
```

Output at end:

```
Summary: 12 matched, 12 run, 0 failed
```

New in v1.3

33. Expression Filter (`--expr`)

The `-f` / `--expr` option lets you compose complex match logic in a single argument, combining `glob()`, `regex()`, and `regex_match()` with boolean operators `and`, `or`, `not`, and parentheses. When `--expr` is used, the third positional argument (regex) is **not required**.

Grammar:

Element	Description
`glob("pattern")`	Convert the glob to an anchored regex and apply `regex_search`
`regex("pattern")`	Substring regex search (default mode)
`regex_search("pattern")`	Alias for `regex()`
`regex_match("pattern")`	Full-path regex match
`and`	Both sides must match
`or`	Either side must match
`not`	Negate the following expression
`( ... )`	Group sub-expressions

Operator precedence (highest to lowest): `not`, `and`, `or`. Use parentheses to override.

Match C++ files, exclude build directories:

```
shell-cmd . "echo %1" --expr '(glob("*.cpp") or glob("*.hpp")) and not regex("build|CMakeFiles")'
```

Single function — equivalent to a regex positional argument:

```
shell-cmd . "wc -l %1" --expr 'regex("\.py$")'
```

Nested boolean logic:

```
shell-cmd . "echo %1" --expr '(glob("*.py") or glob("*.rs")) and not glob("*test*") and not regex("vend
```

Full-path matching inside an expression:

```
shell-cmd . "echo %1" --expr 'regex_match("\\./src/.*\\.cpp")'
```

Combine `--expr` with other options:

```
shell-cmd -x "node_modules" --size +1K --type f . "wc -l %1" --expr 'glob("*.ts") or glob("*.tsx")'
```

Placeholder Quick Reference

Placeholder	Value
`%0`	Filename only (e.g., `report.txt`); in `--list-all` mode, all matched paths joined by spaces
`%1`	Full path (e.g., `/home/user/docs/report.txt`)
`%b`	Basename without extension (e.g., `report`)
`%e`	File extension with dot (e.g., `.txt`)
`%2`	First extra argument after the regex
`%3`	Second extra argument after the regex
`%N`	Nth extra argument (no upper limit)

---

shell-cmd vs `find -exec`

If you're familiar with `find . -exec`, here's how `shell-cmd` compares:

Feature	`shell-cmd`	`find -exec`
Filename placeholder	`%0` gives the filename without the path	No equivalent — requires `sh -c` + `basename`
Full path placeholder	`%1`	`{}`
Extra arguments	`%2`, `%3`, ... with validation	Not supported — use shell variables
Pattern matching	ECMAScript regex on the full path	Glob (`-name`) or implementation-varying `-regex`
Dry-run	Built-in `-n` flag	No native support
Verbose mode	Built-in `-v` flag	No native support
Filtering by metadata	Size (`-s`/`--size`), time (`-m`/`--mtime`), permissions (`-p`/`--perm`), owner (`-u`/`--user`), group (`-g`/`--group`), type (`-t`/`--type`)	Size, time, permissions, ownership, type, boolean logic
Exclude patterns	Built-in `-x` / `--exclude` with regex	Requires negation logic or `! -name`
Parallel execution	Built-in `-j N` / `--jobs N`	Requires `xargs -P` or GNU `parallel`
List-all mode	Built-in `-l` / `--list-all` — run command once with all matches	Requires `xargs` or `+` terminator
Confirm mode	Built-in `-c` / `--confirm`	Requires `-ok` (not universally supported)
Stop on error	Built-in `-e` / `--stop-on-error`	No native support
Summary statistics	Automatic (matched/run/failed counts)	No native support
Portability	Requires C++20 build	POSIX-standard, available everywhere

Side-by-Side Examples

Copy all `.txt` files to a backup directory, preserving filenames:

```
# shell-cmd
shell-cmd . "cp %1 /tmp/backup/%0" ".*\.txt$"
```

```
# find equivalent (needs sh -c + basename gymnastics)
find . -regex '.*\.txt$' -exec sh -c 'cp "$1" "/tmp/backup/$(basename "$1")"' _ {} \;
```

Dry-run to preview commands:

```
# shell-cmd - built-in
shell-cmd -n . "rm %1" ".*\.bak$"

# find - no native dry-run, must rework the command
find . -regex '.*\.bak$' -exec echo rm {} \;
```

Copy files with an extra destination argument:

```
# shell-cmd - %2 is injected and validated
shell-cmd ~/Music "cp %1 %2/%0" ".*\.mp3$" /mnt/backup/music

# find - must hardcode or use a variable
DEST=/mnt/backup/music find ~/Music -regex '.*\.mp3$' -exec sh -c 'cp "$1" "$DEST/$(basename "$1")"' _
```

When to Use Which

- **Use `shell-cmd`** when your command needs the filename separated from the path, when you want to inject extra arguments, or when you want built-in dry-run/verbose, parallel execution, confirm mode, exclude patterns, stop-on-error, composable expression filters (`--expr`), and summary statistics.
- **Use `find`** when you're on a system where you can't compile C++20 code.

Tips and Best Practices

1. **Always dry-run first.** Use `-n` before running destructive commands (`rm`, `mv`, overwriting files) to verify what will execute.
2. **Quote the command template.** Since it contains `%` placeholders and often shell metacharacters, always wrap it in double quotes: `"command %1"`.
3. **Escape regex special characters.** The regex uses ECMAScript syntax. To match a literal dot in file extensions, use `\.` — e.g., `".*\.cpp$"` not `".*cpp$"` (the latter also matches `acpp`). Alternatively, use `--glob` to avoid regex escaping altogether: `--glob "*.cpp"`.
4. **Use `%0` for output filenames.** When copying/converting files to a new directory, `%0` gives you the original filename without the source path, which is ideal for naming outputs.
5. **Combine `-v` with normal execution** to watch progress on long-running batch jobs without doing a separate dry-run pass.
6. **Depth control is useful for large trees.** If you only want files in `src/` and its immediate children, use `-d 1`.
7. **Use metadata filters to narrow results.** Combine `-s`, `-m`, `-p`, `-u`, `-g`, and `-t` to precisely target files without grepping through everything.
8. **Use long-form flags in scripts.** `--dry-run --size +10M --type f` is more readable than `-n -s +10M -t f` when writing reusable shell scripts.
9. **Use `-x` to skip noisy directories.** `-x "node_modules|\.git|build"` saves time and avoids false matches.
10. **Use `-j` for CPU-bound batch work.** Parallel execution shines for tasks like image conversion or compilation where each command is independent.

11. **Use `-c` for irreversible operations.** Confirm mode gives you a per-file safety net when `rm`-ing or `mv`-ing.

Error Handling

Scenario	Behavior
Fewer than 3 positional arguments	Prints error + help text, exits
Directory can't be opened	Prints error with path, exits
Extra argument has no matching placeholder	Prints error naming the missing `%N`, exits
Command fails (non-zero exit)	Next file is processed unless `-e` / `--stop-on-error` is set
Stop-on-error triggered	Processing halts, summary prints, exits with `EXIT_FAILURE`
Invalid regex	`std::regex` throws an exception, program crashes with an unhandled exception

---

License

`shell-cmd` is released under the **GNU GPL v3**. See the LICENSE file for details.