LINUX TEXT PROCESSING

Learning SED

A complete 12-chapter course — from first commands to real-world scripting

CONTENTS

Generated for osztromok.com

CHAPTER 01

Introduction to SED

Chapter 1 — Introduction to SED

What Is SED?

`sed` stands for **S**tream **ED**itor. Unlike a conventional text editor where you open a file, move a cursor around, and make interactive changes, SED processes text as a stream — it reads input line by line, applies a set of instructions to each line, and writes the result to standard output. You never "open" a file in the traditional sense; you pass text through a pipeline of transformations.

SED was written by Lee McMahon at Bell Labs in 1973–74, based directly on the earlier `ed` line editor. It became a standard part of Unix and has been included in every Unix-like operating system since. Today you will find it on Linux, macOS, BSD, and anywhere else that follows the POSIX standard. It is one of the oldest tools in the Unix toolchain that is still in daily use by millions of developers and sysadmins worldwide.

The key insight that makes SED powerful is this: **most text-processing tasks reduce to transformations on a stream of lines**. Finding and replacing text, deleting certain lines, extracting specific sections, reformatting data — SED handles all of these with a compact command syntax that can be expressed in a single shell command or a small script file.

> **SED's place in the Unix philosophy:** The Unix philosophy encourages building small, focused tools that do one thing well and can be combined with pipes. SED embodies this perfectly — it reads from standard input (or a file), transforms text according to your instructions, and writes to standard output. This means SED composes naturally with `grep`, `awk`, `cut`, `sort`, and every other Unix text tool.

Where SED Fits in the Toolchain

Before diving into syntax, it helps to understand when to reach for SED versus its close relatives:

Tool	Primary strength	Typical use
`grep`	Searching — finding lines that match a pattern	Filtering output; checking if a pattern exists
`sed`	Transforming — editing text as it flows through	Find-and-replace; delete/insert lines; restructure text
`awk`	Processing — treating text as structured records with fields	Column arithmetic; reports; complex field-based logic
`cut`	Extracting — slicing out specific columns or fields	Pulling specific fields from CSV or fixed-width data
`tr`	Translating — character-by-character substitution	Case conversion; stripping characters

SED is the right choice when you need to *modify* text — change words, rearrange lines, remove sections, or insert new content — especially when that modification needs to be applied consistently across a large file or integrated into an automated script.

Basic Syntax

Every SED command follows the same fundamental structure:

```
sed [options] 'script' file
```

Breaking that down:

- **`sed`** — the command itself
- **`[options]`** — optional flags that modify SED's behaviour (covered below)
- **`'script'`** — one or more SED commands telling it what to do, enclosed in single quotes
- **`file`** — the input file; if omitted, SED reads from standard input

The simplest SED script is a single command. Here is the classic substitution — replacing one word with another:

```
sed 's/hello/goodbye/' greetings.txt
```

This reads every line of `greetings.txt`, replaces the first occurrence of `hello` with `goodbye` on each line, and prints the result. The original file is unchanged. The output goes to your terminal (standard output).

You can also pipe input directly into SED:

```
echo "hello world" | sed 's/hello/goodbye/'
goodbye world
```

How SED Processes Text — The Cycle

Understanding SED's internal processing cycle is the key to understanding everything else about it. SED works in a loop — one iteration per line of input:

1. **Read** — SED reads one line from the input (stripping the trailing newline) and places it in the **pattern space**
2. **Execute** — SED runs all commands in the script against the pattern space, modifying it as instructed
3. **Print** — unless told otherwise, SED prints the contents of the pattern space to standard output, followed by a newline
4. **Clear** — the pattern space is cleared and the cycle repeats with the next line

The **pattern space** is SED's working buffer — a temporary area in memory where the current line lives while SED processes it. Most SED commands read from and write to the pattern space.

There is also a second buffer called the **hold space** — a persistent area that survives between cycles. This lets you save content from one line and use it when processing a later line. The hold space starts empty and is covered in depth in Chapter 6.

> **Mental model:** Think of SED as a conveyor belt in a factory. Each line of your file is a part that arrives on the belt (pattern space). SED's commands are tools positioned along the belt — they may reshape, stamp, or remove the part. At the end of the belt, the finished part drops into the output. The hold space is a shelf beside the belt where you can temporarily set a part aside.

Your First SED Commands

Substitution — `s`

The substitute command `s` is what most people use SED for. Its syntax is:

```
s/pattern/replacement/flags
```

For now, the basic form without flags replaces the *first* match on each line:

```
# Replace first occurrence of "cat" with "dog" on each line
echo "the cat sat on the cat mat" | sed 's/cat/dog/'
the dog sat on the cat mat

# Add the g (global) flag to replace ALL occurrences on each line
echo "the cat sat on the cat mat" | sed 's/cat/dog/g'
the dog sat on the dog mat
```

Substitution is covered thoroughly in Chapter 2 — for now, just get comfortable with the basic form.

Delete — `d`

The delete command removes lines from the output entirely. Combine it with an address (a line number or pattern) to target specific lines:

```
# Delete line 3
sed '3d' file.txt

# Delete every line containing "DEBUG"
sed '/DEBUG/d' logfile.txt

# Delete blank lines (lines that match the empty-line pattern)
sed '/^$/d' file.txt
```

Print — `p`

The print command explicitly prints the pattern space. On its own this causes each line to be printed twice (once by `p`, once by SED's automatic print at end of cycle). It becomes useful with the `-n` option, which suppresses the automatic print:

```
# Print only lines containing "ERROR" (suppress all others)
sed -n '/ERROR/p' logfile.txt

# Print only lines 5 through 10
sed -n '5,10p' file.txt
```

Common Options

Option	Meaning	Example
`-n`	Suppress automatic printing of the pattern space. Only lines explicitly printed with `p` appear in output.	`sed -n '5p' file.txt`
`-e`	Specify a script expression. Allows multiple commands on the command line.	`sed -e 's/foo/bar/' -e 's/baz/qux/'`
`-f`	Read the script from a file instead of the command line.	`sed -f myscript.sed file.txt`
`-i`	Edit the file **in-place** — write changes back to the original file.	`sed -i 's/old/new/g' file.txt`
`-E` (or `-r`)	Use extended regular expressions (ERE) instead of basic (BRE). Enables `+`, `?`, `()`, `|` without backslashes.	`sed -E 's/(foo|bar)/baz/g'`

In-Place Editing with `-i`

By default SED never touches your input file — it only writes to standard output. The `-i` option changes this, making SED modify the file directly. This is one of the most commonly used options in shell scripts.

```
# Replace "version=1.0" with "version=2.0" inside config.txt
sed -i 's/version=1.0/version=2.0/' config.txt
```

Always make a backup before using `-i` on important files. SED will overwrite the file immediately with no confirmation. Many scripts use `-i.bak` to create a backup automatically — SED saves the original file with the `.bak` extension before writing changes:

```
sed -i.bak 's/old/new/g' important.conf
# Original saved as: important.conf.bak
# Modified version:  important.conf
```

Note: On macOS (BSD sed), a backup suffix with `-i` is required — `-i ''` means in-place with no backup. On GNU/Linux sed, `-i` alone works without a suffix.

Reading From Standard Input vs Files

SED is equally happy reading from a file argument or from a pipe. This flexibility is what makes it so composable with other tools.

```
# From a file
sed 's/foo/bar/' input.txt

# From a pipe
cat input.txt | sed 's/foo/bar/'

# From another command's output
grep "ERROR" logfile.txt | sed 's/ERROR/FAULT/'

# From here-string
sed 's/hello/hi/' <<< "hello world"
hi world

# Multiple files — SED processes them as one continuous stream
sed 's/foo/bar/' file1.txt file2.txt file3.txt
```

When multiple files are given, SED treats them as a single concatenated input. Line numbers are counted continuously across all files.

Running Multiple Commands

SED can run more than one command in a single invocation. There are three ways to do this:

Semicolons (for simple cases)

```
# Two substitutions separated by a semicolon
sed 's/foo/bar/; s/baz/qux/' file.txt
```

Multiple `-e` flags

```
# Each -e introduces one command
sed -e 's/foo/bar/' -e 's/baz/qux/' -e '/DEBUG/d' file.txt
```

A script file with `-f`

```
# Write commands into a file, one per line
# Contents of cleanup.sed:
#   s/foo/bar/g
#   s/baz/qux/g
#   /^$/d
sed -f cleanup.sed file.txt
```

Script files are the cleanest option for anything longer than two or three commands. They are easier to read, easier to maintain, and can be version-controlled like any other code.

GNU SED vs BSD SED

There are two main flavours of SED you will encounter:

Flavour	Found on	Notable differences
GNU sed	Linux (most distributions)	Richer feature set; `-i` works without suffix; supports `\w`, `\+`, `\?` in BRE; `-E` or `-r` for ERE
BSD sed	macOS, FreeBSD, OpenBSD	Stricter POSIX compliance; `-i ''` required (empty suffix); `-E` for ERE (not `-r`)

Throughout this course, examples use GNU SED conventions (Linux). Where a macOS difference matters, it will be noted. If you are on macOS and want GNU SED behaviour, you can install it with Homebrew: `brew install gnu-sed` — it installs as `gsed`.

> **Check your version:** Run `sed --version` on Linux to see which version of GNU sed you have. On macOS, `sed --version` will print an error (BSD sed does not support it); try `sed --help 2>&1 | head -3` instead, or simply check `man sed` for the BSD-specific documentation.

A Practical First Example

Let us put the basics together with a realistic scenario. Suppose you have a configuration file and you want to update a setting without opening an editor:

```
# app.conf — before
debug=false
log_level=warn
max_connections=10
server_name=localhost
```

```
# Change log_level from warn to info, and increase max_connections to 50
sed -i.bak -e 's/log_level=warn/log_level=info/' \
           -e 's/max_connections=10/max_connections=50/' app.conf

# app.conf — after
debug=false
log_level=info
max_connections=50
server_name=localhost
```

The original is preserved as `app.conf.bak`. The changes are applied atomically in a single SED pass — the file is read once, both substitutions run on each line, and the result is written back.

Quick Reference — Chapter 1

CORE SYNTAX

`sed 'cmd' file`	Run a SED command against a file; output goes to stdout
`cmd | sed 'cmd'`	Read input from a pipe
`sed -e 'cmd1' -e 'cmd2'`	Run multiple commands on the command line
`sed -f script.sed file`	Read commands from a script file

COMMON OPTIONS

`-n`	Suppress automatic printing; only print when explicitly told to with `p`
`-i`	Edit file in-place (overwrites original)
`-i.bak`	Edit in-place, saving original with `.bak` extension
`-E`	Enable extended regular expressions (ERE)

INTRODUCTORY COMMANDS

`s/pattern/replacement/`	Substitute — replace first match of *pattern* with *replacement* on each line
`s/pattern/replacement/g`	Substitute all matches on each line (global flag)
`d`	Delete the current line (do not print it)
`p`	Print the pattern space explicitly

What is coming next: Chapter 2 digs deep into the substitute command — the single most important SED command. You will learn every flag, how to use capture groups and backreferences, how to change the delimiter when your pattern contains slashes, and a library of practical substitution examples.

CHAPTER 02

The Substitute Command

Chapter 2 — The Substitute Command

The Most Important Command in SED

If you only ever learn one SED command, make it `s` — the substitute command. It is by far the most commonly used, and mastering it covers the vast majority of real-world SED use cases. This chapter is a complete guide to everything `s` can do.

The full syntax of the substitute command is:

```
s/pattern/replacement/flags
```

Each part has a precise role:

- `s` — the command itself (substitute)
- `/` — delimiter; separates the three parts (can be changed — see below)
- `pattern` — a regular expression to search for in the pattern space
- `replacement` — the text to substitute in place of the matched portion
- `flags` — optional modifiers controlling how the substitution behaves

SED applies the pattern against the entire pattern space (the current line). When it finds a match, it replaces the matched text with the replacement. Everything outside the matched portion is left exactly as it was.

Substitution Flags

Flags appear after the final delimiter and modify how the substitution operates. Multiple flags can be combined.

The `g` flag — Global replacement

Without `g`, SED replaces only the *first* match on each line. The `g` flag replaces *all* matches:

```
# Without g — only the first "the" is replaced
echo "the cat and the dog and the bird" | sed 's/the/a/'
a cat and the dog and the bird

# With g — all occurrences replaced
echo "the cat and the dog and the bird" | sed 's/the/a/g'
a cat and a dog and a bird
```

The `i` flag — Case-insensitive matching (GNU sed)

Makes the pattern match regardless of case:

```
echo "Error ERROR error eRrOr" | sed 's/error/fault/gi'
fault fault fault fault
```

> **POSIX note:** The `i` flag is a GNU sed extension and is not available in strictly POSIX-compliant or BSD sed. For portable scripts, use a character class instead: `s/[Ee][Rr][Rr][Oo][Rr]/fault/g`.

The `N` th occurrence flag — Replace a specific occurrence

A numeric flag tells SED which occurrence to replace. `2` means the second match, `3` the third, and so on:

```
echo "aa bb aa cc aa dd" | sed 's/aa/XX/2'
aa bb XX cc aa dd

# Combine with g to replace from the Nth occurrence onwards
echo "aa bb aa cc aa dd" | sed 's/aa/XX/2g'
aa bb XX cc XX dd
```

The `p` flag — Print if substitution was made

Prints the pattern space if and only if the substitution succeeded. Most useful with `-n` to show only lines that were actually changed:

```
# Show only lines where a substitution was made
sed -n 's/foo/bar/p' file.txt

# Practical: show which config lines contain "debug" and what they changed to
sed -n 's/debug=true/debug=false/p' app.conf
```

The `w` flag — Write matching lines to a file

Writes the pattern space to a named file whenever the substitution succeeds:

```
# Apply substitution AND write changed lines to changes.log
sed 's/ERROR/FAULT/w changes.log' logfile.txt
```

The `e` flag — Execute replacement as a shell command (GNU sed)

After substitution, the resulting pattern space is executed as a shell command and the output replaces the line. Powerful but use carefully:

```
# Get the file size of each filename listed in a file
sed 's/.*/du -sh & 2>\/dev\/null/' filelist.txt | sed -e 's/.*/&/'e
```

> **The `e` flag executes arbitrary shell commands.** Never use it on untrusted input — it is a remote code execution risk if the input can be controlled by an attacker.

Flag summary table

Flag	Effect	Portable?
`g`	Replace all matches on the line (not just the first)	Yes — POSIX
`1`, `2`, `N`	Replace only the Nth occurrence	Yes — POSIX
`Ng`	Replace from the Nth occurrence onwards	GNU sed only
`p`	Print pattern space if substitution succeeded	Yes — POSIX
`w file`	Write to *file* if substitution succeeded	Yes — POSIX
`i`	Case-insensitive match	GNU sed only
`e`	Execute result as shell command	GNU sed only
`m`	Multi-line mode — `^` / `$` match embedded newlines	GNU sed only

Changing the Delimiter

The `/` character is SED's default delimiter, but it is just a convention. SED accepts any character immediately after `s` as the delimiter. This matters most when your pattern or replacement *contains* forward slashes — otherwise you would need to escape every one with a backslash.

Compare these two equivalent commands:

```
# Using / as delimiter — slashes in the path must be escaped
sed 's/\/etc\/nginx\/nginx.conf/\/etc\/nginx\/nginx.conf.bak/' file.txt

# Using | as delimiter — much more readable
sed 's|/etc/nginx/nginx.conf|/etc/nginx/nginx.conf.bak|' file.txt

# Using # as delimiter — common convention for path substitutions
sed 's#/usr/local/bin#/usr/bin#g' scripts.txt

# Using , as delimiter — works with URLs too
sed 's,https://old-domain.com,https://new-domain.com,g' links.html
```

> **Convention:** Use `|` or `#` when working with file paths, URLs, or any text containing forward slashes. Your scripts will be much easier to read.

Special Replacement Sequences

The replacement string is not plain text — it has its own set of special sequences that expand to useful values at substitution time.

`&` — The entire matched text

`&` in the replacement expands to whatever the pattern matched. This lets you wrap, prefix, or suffix the matched text without re-typing it:

```
# Wrap every number in brackets
echo "port 8080 and timeout 30" | sed 's/[0-9][0-9]*/[&]/g'
port [8080] and timeout [30]

# Quote every word
echo "alpha beta gamma" | sed 's/[a-z]*/"\0&"/g'

# Add a prefix to every line matching a pattern
sed 's/^ERROR.*/[ALERT] &/' logfile.txt
# Before: ERROR connection refused
# After:  [ALERT] ERROR connection refused
```

`\1`, `\2`, ... `\9` — Capture group backreferences

When your pattern contains groups enclosed in `\(` and `\)` (in basic regex, BRE), each group's matched text is captured. `\1` in the replacement refers to the first group, `\2` to the second, and so on. This is one of the most powerful features of the substitute command.

```
# Swap the order of first and last name (BRE groups)
echo "Smith, John" | sed 's/\([^,]*\), \(.*\)/\2 \1/'
John Smith

# With -E (ERE) the groups are written without backslashes
echo "Smith, John" | sed -E 's/([^,]*), (.*)/\2 \1/'
John Smith

# Reformat a date from YYYY-MM-DD to DD/MM/YYYY
echo "2024-07-15" | sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/'
15/07/2024

# Extract the domain from a URL
echo "https://www.example.com/path/page" | sed -E 's|https?://([^/]+).*|\1|'
www.example.com
```

Case modifiers (GNU sed only)

GNU sed supports special escape sequences in the replacement to change the case of matched text:

Sequence	Effect
`\u`	Make the next character uppercase
`\l`	Make the next character lowercase
`\U`	Make all following characters uppercase (until `\E` or end)
`\L`	Make all following characters lowercase (until `\E` or end)
`\E`	End a `\U` or `\L` transformation

```
# Capitalise the first letter of every word
echo "hello world from sed" | sed 's/\b\([a-z]\)/\u\1/g'
Hello World From Sed

# Convert matched text to uppercase
echo "warning: disk space low" | sed 's/warning/\U&/'
WARNING: disk space low

# Convert entire line to lowercase
echo "SYSTEM ALERT: DISK FULL" | sed 's/.*/\L&/'
system alert: disk full
```

Escaping in Patterns and Replacements

Some characters have special meaning in SED patterns or replacements and must be escaped with a backslash when you want them treated literally:

Character	Special meaning	Escape as	Context
`/`	Delimiter (default)	`\/` or change delimiter	Pattern and replacement
`&`	The entire match	`\&`	Replacement only
`\`	Escape character	`\\`	Pattern and replacement
`.`	Match any character	`\.`	Pattern only
`*`	Zero or more of preceding	`\*`	Pattern only
`^`	Start of line	`\^`	Pattern only
`$`	End of line	`\$`	Pattern only
`[`	Start of character class	`\[`	Pattern only
`\n`	Newline character	literal (in replacement)	Both

```
# Replace a literal dot (not "any character")
echo "version 1.0.5" | sed 's/1\.0\.5/2\.0\.0/'
version 2.0.0

# Insert a literal ampersand into the replacement
echo "Tom Jerry" | sed 's/ / \& /'
Tom & Jerry

# Replace a literal backslash
echo 'C:\Users\emuba' | sed 's/\\/\//g'
C:/Users/emuba
```

Inserting Newlines

You can insert a literal newline into the replacement string. In GNU sed, use `\n` in the replacement:

```
# Split a comma-separated pair onto two lines
echo "name:John,age:30" | sed 's/,/\n/g'
name:John
age:30

# Add a blank line before every section heading
sed 's/^\[.*\]/\n&/' config.ini
```

To insert a newline in a portable (POSIX) way, use a backslash followed by a literal newline inside the script:

```
# POSIX-portable newline in replacement (note the literal newline after \)
sed 's/,/\
/g' file.txt
```

Addresses with Substitution

By default the substitute command runs on every line. You can restrict it to specific lines by prefixing an *address* — a line number or a pattern. Addresses are covered fully in Chapter 3, but here is a preview because they are so commonly combined with substitution:

```
# Only substitute on line 5
sed '5s/old/new/' file.txt

# Only substitute on lines 10 through 20
sed '10,20s/old/new/g' file.txt

# Only substitute on lines containing "SECTION"
sed '/SECTION/s/old/new/' file.txt

# Only substitute on lines NOT containing "SECTION" (! negates the address)
sed '/SECTION/!s/old/new/' file.txt

# Between a start pattern and an end pattern
sed '/START/,/END/s/foo/bar/g' file.txt
```

Practical Recipes

Here are real-world substitution patterns you will use regularly:

Trim leading and trailing whitespace

```
# Trim leading whitespace (spaces and tabs)
sed 's/^[[:space:]]*//' file.txt

# Trim trailing whitespace
sed 's/[[:space:]]*$//' file.txt

# Trim both in one pass
sed 's/^[[:space:]]*//; s/[[:space:]]*$//' file.txt
```

Comment out a line in a config file

```
# Add a # to the beginning of a line containing "ServerName"
sed 's/^\(ServerName\)/#\1/' httpd.conf

# Uncomment a line (remove leading #)
sed 's/^#\(ServerName\)/\1/' httpd.conf
```

Update a version number

```
# Replace semantic version (flexible — matches any x.y.z)
sed -E 's/version = "[0-9]+\.[0-9]+\.[0-9]+"/version = "2.1.0"/' pyproject.toml

# Increment the patch version (extract groups, add 1 — requires awk for arithmetic)
sed -E 's/(version = ")([0-9]+\.[0-9]+\.)([0-9]+)(")/\1\2PATCH\4/' file.txt
```

Remove HTML tags

```
# Strip all HTML/XML tags from a file
sed 's/<[^>]*>//g' page.html

# Remove a specific tag and its closing tag
sed 's/<b>//g; s/<\/b>//g' page.html
```

Reformat log timestamps

```
# Change 2024-07-15 12:30:00 to 15/07/2024 12:30:00
sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/g' app.log
```

Mask sensitive data

```
# Replace credit card numbers (16 digits) with asterisks
sed -E 's/[0-9]{4}[ -]?[0-9]{4}[ -]?[0-9]{4}[ -]?[0-9]{4}/****-****-****-****/g' data.txt

# Mask everything after "password=" keeping the key name
sed -E 's/(password=).*/\1[REDACTED]/' config.txt
```

Normalise line endings

```
# Convert Windows CRLF (\r\n) to Unix LF (\n)
sed 's/\r//' windows_file.txt

# Same with -i to fix the file in place
sed -i 's/\r//' windows_file.txt
```

Replace across multiple files at once

```
# Update an API endpoint URL across all .js files in a project
sed -i 's|https://api.old-domain.com|https://api.new-domain.com|g' $(find src/ -name "*.js")

# Safer: preview changes first before -i (dry run using diff)
sed 's|https://api.old-domain.com|https://api.new-domain.com|g' app.js | diff app.js -
```

BRE vs ERE — A Quick Reminder

The substitute command uses **Basic Regular Expressions (BRE)** by default. With the `-E` flag it uses **Extended Regular Expressions (ERE)**. The practical difference is how you write grouping, alternation, and quantifiers:

Feature	BRE (default)	ERE (with -E)
Grouping	`\(` and `\)`	`(` and `)`
One or more	`\+` (GNU) or `[a-z][a-z]*`	`+`
Zero or one	`\?` (GNU) or workaround	`?`
Alternation	`|` (GNU) or not supported	`|`
Repetition	`\{3\}`, `\{2,5\}`	`{3}`, `{2,5}`
Backreferences	`\1` to `\9`	`\1` to `\9`

> **Recommendation:** Use `-E` whenever your pattern involves groups, quantifiers like `+` or `?`, or alternation. ERE syntax is cleaner and less error-prone than BRE. Only reach for BRE when writing POSIX-portable scripts that must run on BSD sed without `-E` support.

Common Mistakes

Greedy matching going too far

```
# Trying to remove the content inside <b> tags on a single line
echo "<b>bold</b> and <b>more bold</b>" | sed 's/<b>.*<\/b>//'
# Removes EVERYTHING between first <b> and last </b> — greedy!
 and   # probably not what you wanted

# Fix: use [^<]* instead of .* to stop at the first tag character
echo "<b>bold</b> and <b>more bold</b>" | sed 's/<b>[^<]*<\/b>//g'
 and   # hmm — still empty; let's see the full output below

echo "before <b>bold</b> middle <b>more</b> after" | sed 's/<b>[^<]*<\/b>//g'
before  middle  after
```

Forgetting that `.` matches anything — including dots

```
# Trying to match a literal IP-style string
echo "192.168.1.1 and 192X168Y1Z1" | sed 's/192.168.1.1/REDACTED/g'
# Both match because . means "any character"!
REDACTED and REDACTED

# Fix: escape the dots
echo "192.168.1.1 and 192X168Y1Z1" | sed 's/192\.168\.1\.1/REDACTED/g'
REDACTED and 192X168Y1Z1
```

Applying `-i` without testing first

```
# ALWAYS test without -i first, then add it when satisfied
# Step 1: preview — does the output look right?
sed 's/old_value/new_value/g' important.conf

# Step 2: only then edit in place with a backup
sed -i.bak 's/old_value/new_value/g' important.conf
```

Quick Reference — Chapter 2

SUBSTITUTE COMMAND SYNTAX

`s/pattern/replacement/`	Replace first match of *pattern* with *replacement*
`s/pattern/replacement/g`	Replace all matches on the line
`s/pattern/replacement/2`	Replace only the second match
`s/pattern/replacement/2g`	Replace from the second match onwards
`s/pattern/replacement/i`	Case-insensitive match (GNU sed)
`s/pattern/replacement/p`	Print pattern space if substitution succeeded
`s/pattern/replacement/w file`	Write to *file* if substitution succeeded

SPECIAL REPLACEMENT SEQUENCES

`&`	The entire matched text
`\1 … \9`	Captured group 1 through 9
`\n`	Newline (GNU sed)
`\u \l \U \L \E`	Case conversion (GNU sed)

DELIMITER ALTERNATIVES

`s|pattern|replacement|`	Use | as delimiter — good for paths/URLs
`s#pattern#replacement#`	Use # as delimiter
`s,pattern,replacement,`	Use , as delimiter

> **What is coming next:** Chapter 3 covers addresses — the mechanism that lets you target specific lines or ranges of lines for any SED command. You have already seen line numbers and pattern addresses briefly; Chapter 3 goes deep into all address types, ranges, step addresses, and negation.

CHAPTER 03

Addresses and Line Selection

Chapter 3 — Addresses and Line Selection

What Is an Address?

Every SED command runs against the pattern space — by default, on *every line* of the input. An **address** is a restriction you place before a command to limit which lines it acts on.

The general form is:

```
sed 'address command' file
```

If you provide no address, the command runs on all lines. If you provide an address, the command only runs when SED is processing a line that matches that address.

SED supports four types of address:

- **Line number addresses** — target a specific line by its position
- **The last-line address (`$`)** — target the final line of input
- **Pattern addresses** — target lines matching a regular expression
- **Range addresses** — target a span of lines between a start and end address

Addresses can be combined with **negation** (`!`) to target every line *except* those that match, and with **step notation** to target every Nth line.

Line Number Addresses

The simplest address is a plain line number. SED counts lines from 1:

```
# Delete line 1 (commonly used to strip a header)
sed '1d' file.txt

# Substitute only on line 5
sed '5s/old/new/' file.txt

# Print only line 3
sed -n '3p' file.txt
```

```
sed '3s/foo/bar/' — line 3 address
```

```
line 1: apple foo tree ← skipped (not line 3) line 2: banana foo bush ← skipped line 3: cherry foo plant ←
ADDRESS MATCHES → s/foo/bar/ runs → "cherry bar plant" line 4: date foo tree ← skipped line 5: elderberry
foo ← skipped
```

The Last-Line Address: `$`

`$` matches the very last line of the input. Because SED processes streams, it does not know in advance how many lines there are — `$` is resolved dynamically as SED reads:

```
# Delete the last line
sed '$d' file.txt

# Append text after the last line
sed '$a\--- end of file ---' file.txt

# Substitute only on the last line
sed '$s/old/new/' file.txt

# Print only the last line (equivalent to tail -1)
sed -n '$p' file.txt
```

> **Do not confuse `$` the line address with `$` the regex anchor.** Inside a pattern address (`/regex/`), `$` means end-of-line. As a standalone address before a command, `$` means the last line of the file. Context tells SED which meaning applies.

Pattern Addresses

A pattern address matches any line where a regular expression is found. Enclose the regex in forward slashes:

```
# Delete all lines containing "DEBUG"
sed '/DEBUG/d' logfile.txt

# Substitute only on lines containing "ERROR"
sed '/ERROR/s/server1/server2/' logfile.txt

# Print only lines starting with a digit
sed -n '/^[0-9]/p' file.txt

# Delete blank lines
sed '/^$/d' file.txt

# Delete lines containing only whitespace
sed '/^[[:space:]]*$/d' file.txt
```

```
sed '/ERROR/d' — pattern address
```

```
2024-01-15 INFO server started ← no match, printed as-is 2024-01-15 ERROR connection refused ← MATCHES
/ERROR/ → d runs → line deleted 2024-01-15 INFO request received ← no match, printed as-is 2024-01-15 ERROR
timeout after 30s ← MATCHES /ERROR/ → d runs → line deleted 2024-01-15 INFO server stopped ← no match,
printed as-is
```

Changing the pattern address delimiter

Just like the substitute command, pattern addresses can use an alternative delimiter. This is essential when the regex itself contains forward slashes (such as file paths):

```
# Delete lines containing a file path — awkward with /
sed '/\/usr\/local\/bin/d' file.txt

# Same thing using \| as alternate delimiter — much cleaner
sed '\|/usr/local/bin|d' file.txt

# Using # as delimiter
sed '\#/etc/nginx#d' file.txt
```

> **Syntax for alternate pattern delimiters:** Lead with a backslash, then your chosen delimiter character, then the regex, then the delimiter again: `\delimregexdelim`. This is different from the substitute command where you just start with the delimiter directly after `s`.

Range Addresses

A range address targets every line between a start address and an end address (inclusive). Start and end are separated by a comma:

```
start,end command
```

Both start and end can independently be a line number, `$`, or a pattern address — and they can be mixed:

Line number ranges

```
# Delete lines 5 through 10 (inclusive)
sed '5,10d' file.txt

# Print lines 20 to 30
sed -n '20,30p' file.txt

# Substitute from line 3 to end of file
sed '3,$s/old/new/g' file.txt
```

Pattern ranges

When both addresses are patterns, SED activates the range when the start pattern first matches, and deactivates it when the end pattern next matches. Lines between those two — including the matching lines themselves — are all addressed:

```
# Delete everything between (and including) START and END markers
sed '/START/,/END/d' file.txt

# Substitute inside an XML/HTML block
sed '/<config>/,/<\/config>/s/false/true/g' settings.xml

# Comment out a block of lines in a shell script
sed '/^# BEGIN BLOCK/,/^# END BLOCK/s/^/#/' script.sh
```

```
sed '/START/,/END/d'
line 1: header text ← before range, printed line 2: START marker ← start pattern matches → range opens → deleted line 3: inside the block ← inside range → deleted line 4: still inside ← inside range → deleted line 5: END marker ← end pattern matches → deleted, range closes line 6: footer text ← after range, printed
```

Mixed ranges (line + pattern)

```
# From line 1 up to (and including) the first blank line
sed -n '1,/^$/p' file.txt

# From the line matching "BEGIN" to line 50
sed '/BEGIN/,50d' file.txt

# From line 5 to the line matching "DONE"
sed -n '5,/DONE/p' file.txt
```

The range end-pattern pitfall

> **The end pattern is only checked from the line *after* the start match.** This means if the start pattern and end pattern both match the same line, the range will not close on that line — SED will carry on until the end pattern is found on a *subsequent* line (or until end of file if it never matches again).
>
> ```
> # If START and END appear on the same line, range stays open longer than expected
> printf 'before\nSTART END here\nafter\n' | sed '/START/,/END/d'
> # Output: before
> # "after" is ALSO deleted because the range never closed on the START END line
> ```

Step Addresses

GNU sed extends the addressing syntax with **step addresses**, written as `first~step`. This matches line `first`, then every `step` th line after that:

```
# Match every other line starting from line 1 (odd lines: 1, 3, 5, …)
sed -n '1~2p' file.txt

# Match even lines (2, 4, 6, …)
sed -n '2~2p' file.txt

# Match every 3rd line starting from line 3 (3, 6, 9, …)
sed -n '3~3p' file.txt

# Delete every 5th line (5, 10, 15, …) — e.g. strip blank separator lines
sed '5~5d' file.txt

# Add a separator after every 4th line (useful for visual grouping)
sed '4~4a\---' file.txt
```

```
sed -n '1~2p' — every odd line

line 1 ← 1~2 matches (1) → printed  line 2 ← no match → suppressed  line 3 ← 1~2 matches (1+2) → printed  line 4 ← no match → suppressed  line 5 ← 1~2 matches (1+4) → printed  line 6 ← no match → suppressed
```

> **Step addresses are a GNU sed extension.** They are not available in BSD sed or strictly POSIX environments. To select odd/even lines portably, use the hold space trick (covered in Chapter 6) or `awk 'NR%2==1'`.

Negating an Address with `!`

Appending `!` immediately after an address (before the command) inverts the match — the command runs on every line that does *not* match the address:

```
# Print every line EXCEPT line 1 (skip the header)
sed -n '1!p' file.txt

# Delete every line that does NOT contain "KEEP"
sed '/KEEP/!d' file.txt

# Substitute on every line except the last
sed '$!s/old/new/g' file.txt

# Substitute on every line outside a block
sed '/START/,/END/!s/foo/bar/' file.txt

# Negate a range — act on lines OUTSIDE the range
sed '5,10!s/foo/bar/' file.txt
```

> **Negation + delete = filter.** The pattern `/regex/!d` (delete lines that do *not* match) is equivalent to `grep regex` but processes the file in a SED pipeline — useful when you need the output to feed directly into more SED commands.

Multiple Commands on One Address — Braces

You can apply several commands to the same address by grouping them in curly braces `{ }`. All commands inside the braces run only when the address matches:

```
# On lines containing "ERROR": substitute AND print
sed -n '/ERROR/ {
  s/ERROR/FAULT/
  p
}' logfile.txt

# On lines 5-15: strip leading spaces AND remove trailing punctuation
sed '5,15 {
  s/^[[:space:]]*//
  s/[[:punct:]]*$//
}' file.txt

# Inline with semicolons (same effect, less readable for complex cases)
sed '/ERROR/ { s/ERROR/FAULT/; p; }' logfile.txt
```

Braces can also be nested — an outer address restricts a range, and an inner address further filters within it:

```
# Between START and END markers, only substitute on lines containing "host"
sed '/START/,/END/ {
  /host/ s/server1/server2/
}' config.txt
```

Address Summary Table

Address form	Matches	Example
`n`	Line number *n*	`sed '5d'`
`$`	The last line	`sed '$d'`
`/regex/`	Any line where *regex* matches	`sed '/ERROR/d'`
`\Xregex X`	Pattern with alternate delimiter *X*	`sed '|/usr|d'`
`n,m`	Lines *n* through *m* inclusive	`sed '3,7d'`
`n,$`	Line *n* through end of file	`sed '5,$d'`
`/pat1/,/pat2/`	From first line matching *pat1* to next line matching *pat2*	`sed '/BEGIN/,/END/d'`
`n,/regex/`	From line *n* to next line matching *regex*	`sed '1,/^$/d'`
`first~step`	Line *first*, then every *step*th line (GNU)	`sed '1~2d'`
`addr!`	All lines that do *not* match *addr*	`sed '/KEEP/!d'`
`addr { cmds }`	Multiple commands under one address	`sed '/ERR/{s/E/e/;p}'`

Practical Recipes

Remove a CSV or TSV header row

```
# Skip the first line (header), pass the rest to downstream commands
sed '1d' data.csv | awk -F, '{print $2}'
```

Extract a block between two markers

```
# Print only the lines between (and including) BEGIN and END
sed -n '/BEGIN/,/END/p' file.txt

# Print the block but EXCLUDE the marker lines themselves
sed -n '/BEGIN/,/END/ { /BEGIN/d; /END/d; p }' file.txt
```

Process only a specific function in a script

```
# Add debug echo before every line inside the "deploy" function
sed '/^deploy()/,/^}/ {
  /^[[:space:]]\+[^}]/ s/^/echo "DEBUG: " #/
}' deploy.sh
```

Remove comments and blank lines

```
# Strip # comments and empty lines from a config file
sed '/^[[:space:]]*#/d; /^[[:space:]]*$/d' nginx.conf
```

Operate only outside a quoted string block

```
# Substitute only on lines that do NOT look like SQL string literals
sed "/^'/!s/NULL/0/g" dump.sql
```

Print line numbers alongside matching lines

```
# Equivalent of grep -n "pattern" but done entirely in SED
sed -n '/ERROR/ {=; p}' logfile.txt
# = prints the current line number; p prints the line
42
2024-01-15 ERROR connection refused
```

Apply edits from line N to end of file (skip boilerplate header)

```
# A generated file has a 10-line header that must not be touched
sed '11,$s/v1/v2/g' generated.txt
```

Common Mistakes with Addresses

Confusing line 0 and line 1

```
# SED counts from 1 — there is no line 0
sed '0d' file.txt
# GNU sed: "invalid usage of line address 0"

# Special case: 0,/regex/ — the only place 0 is valid
# It means: start immediately (before reading line 1)
# allowing the end-pattern to match on line 1 itself
sed '0,/pattern/d' file.txt
# Deletes from start up to AND including the first line matching /pattern/
# even if that line is line 1
```

Pattern ranges that never close

```
# If END never appears after START, the range stays open to EOF
printf 'a\nSTART\nb\nc\n' | sed '/START/,/END/d'
a
# Lines b and c are also deleted — the range never closed
# Always test on sample input before running on a real file
```

Forgetting that ranges are re-evaluated

```
# A pattern range matches EVERY occurrence of the start pattern
printf 'START\na\nEND\nSTART\nb\nEND\n' | sed '/START/,/END/d'
# Both blocks are deleted — the range activates twice
# If you only want the FIRST block, use 0,/END/ after the START line
```

Quick Reference — Chapter 3

ADDRESS TYPES

`n`	Line *n* (first line is 1)
`$`	Last line of input
`/regex/`	Any line matching *regex*
`\Xregex X`	Pattern address with alternate delimiter *X*

RANGE AND STEP

`n,m`	Lines *n* through *m*
`/p1/,/p2/`	From line matching *p1* to line matching *p2*
`0,/regex/`	From start to first match (allows match on line 1)
`first~step`	Line *first* then every *step*th line (GNU sed)

MODIFIERS AND GROUPING

`addr!`	Negate — run command on lines that do *not* match address
`addr { cmd1; cmd2 }`	Run multiple commands under one address
`addr1 { addr2 cmd }`	Nested addresses — outer restricts, inner further filters

What is coming next: Chapter 4 covers the delete, print, and quit commands in full — including how `-n` with `p` lets you use SED as a powerful line-extraction tool, and how `q` and `Q` let you stop processing early for efficiency on large files.

CHAPTER 04

Delete, Print and Quit

Chapter 4 — Delete, Print and Quit Commands

Three Commands That Control the Cycle

While the substitute command transforms text, the three commands in this chapter control something more fundamental — *whether* a line appears in the output at all, and *when* SED stops reading. Together with the `-n` flag, they give you precise control over what gets printed and when processing ends.

Command	What it does	Effect on cycle
`d`	Delete the pattern space	Skips the rest of the script and the default print; jumps immediately to the next cycle
`p`	Print the pattern space	Outputs the pattern space now (in addition to the default print at end of cycle)
`q`	Quit after this line	Prints the pattern space (unless `-n`), then exits immediately
`Q`	Quit without printing	Exits immediately without printing the pattern space (GNU sed)

The Delete Command: `d`

The `d` command deletes the pattern space and **immediately starts the next cycle** — it does not just suppress output, it aborts all remaining commands for the current line and jumps straight to reading the next line. No further commands in the script run after `d` fires.

```
Normal cycle (no d): read line → run commands → print → next line
Cycle with d firing: read line → run commands → d fires → skip remaining commands → skip print → next line
```

Basic deletion

```
# Delete every line containing "TODO"
sed '/TODO/d' notes.txt

# Delete blank lines
sed '/^$/d' file.txt

# Delete lines containing only whitespace
sed '/^[[:space:]]*$/d' file.txt

# Delete lines shorter than 3 characters (e.g. stray single-char lines)
sed '/^.\{0,2\}$/d' file.txt

# Delete the first line (strip a header row)
sed '1d' data.csv

# Delete the last line
sed '$d' file.txt

# Delete a range of lines
sed '3,7d' file.txt
```

Deleting comment lines and cleaning config files

```
# Remove shell-style # comments
sed '/^[[:space:]]*#/d' script.sh

# Remove both comments AND blank lines in one pass
sed '/^[[:space:]]*#/d; /^[[:space:]]*$/d' nginx.conf

# Remove inline comments (everything from # to end of line)
sed 's/[[:space:]]*#.*$//' config.txt
# Note: this uses s// not d — it removes the comment but keeps the line
```

Deleting between markers

```
# Delete a block including its opening and closing tags
sed '/<!-- BEGIN SKIP -->/,/<!-- END SKIP -->/d' page.html

# Delete from a pattern to end of file
sed '/^### Appendix/,$d' README.md

# Delete from line 1 to the first blank line (strip email headers)
sed '1,/^$/d' email.txt
```

Why `d` short-circuits the script

Because `d` aborts the rest of the script for that line, the order of your commands matters. A command placed after `d` in the same address block will never run on lines where `d` fires:

```
# The substitution on line 2 will NEVER run for lines matching /DEBUG/
sed '/DEBUG/ {
  d
  s/foo/bar/    # unreachable — d already ended this cycle
}' file.txt

# If you need to transform before deleting, put the transform first
sed '/DEBUG/ {
  s/DEBUG/INFO/   # runs first
  d               # then deletes (so the substitution is pointless here)
}' file.txt
# In practice, if you're deleting the line anyway, transforms before d are useless
```

The Print Command: `p`

The `p` command prints the current contents of the pattern space to standard output. Crucially, SED *also* prints the pattern space automatically at the end of each cycle (the default print). This means that without `-n`, using `p` causes each matching line to appear **twice**.

```
sed 'p' on a line "hello": p fires → prints "hello" ← explicit print end cycle → prints "hello" ← default
print output: hello hello
```

This double-printing behaviour is not a bug — it becomes genuinely useful in specific patterns, and the fix when you don't want it is always `-n`.

The `-n` flag — suppressing the default print

The `-n` flag (sometimes called "silent mode" or "quiet mode") tells SED not to print the pattern space automatically at the end of each cycle. With `-n` active, the *only* output you get is from explicit `p` commands (or the `p` flag on `s`). This turns SED into a powerful line-extraction tool:

```
# Without -n: every line printed, matching lines printed twice
printf 'a\nb\nc\n' | sed '/b/p'
a
b
b
c

# With -n: only explicitly printed lines appear
printf 'a\nb\nc\n' | sed -n '/b/p'
b
```

Using `-n` with `p` for line extraction

```
# Print only line 7
sed -n '7p' file.txt

# Print lines 10 through 20
sed -n '10,20p' file.txt

# Print lines matching a pattern
sed -n '/^ERROR/p' logfile.txt

# Print lines matching either of two patterns (ERE alternation)
sed -En '/(ERROR|WARN)/p' logfile.txt

# Print a block between two markers (inclusive)
sed -n '/BEGIN/,/END/p' file.txt

# Print the last line (equivalent to tail -1)
sed -n '$p' file.txt

# Print every other line (odd lines)
sed -n '1~2p' file.txt
```

Using `p` intentionally without `-n` — showing before and after

Sometimes the double-print behaviour of `p` is deliberate. A classic use is showing what a substitution changed — print the original line with `p` first, then let the substitution and default print show the modified version:

```
# Show original and modified side by side for every changed line
sed '/foo/ { p; s/foo/bar/; }' file.txt
# Lines with "foo":
#   p prints the original    → "this foo line"
#   s replaces it            → "this bar line"
#   default print outputs it → "this bar line"
# Lines without "foo": only appear once (default print)
```

The `p` flag on the substitute command

As seen in Chapter 2, the `p` flag on `s` prints the pattern space when the substitution succeeds. Combined with `-n`, this is the cleanest way to see only lines that were actually changed:

```
# Print only lines where a substitution was made
sed -n 's/foo/bar/p' file.txt

# Practical: check which config values will change before committing with -i
sed -n 's/debug=true/debug=false/p' app.conf
```

The Line Number Command: `=`

Though not strictly a print variant, the `=` command is closely related — it prints the current line number to standard output, followed by a newline. It is worth covering here because it is most useful alongside `p`:

```
# Print line numbers for every line (like cat -n)
sed '=' file.txt | paste - -
# paste merges the number line and content line onto one line

# Print line numbers AND lines for lines matching a pattern (like grep -n)
sed -n '/ERROR/ { =; p }' logfile.txt
42
2024-01-15 ERROR connection refused
87
2024-01-15 ERROR disk full

# Count total lines in a file (print only the last line number)
sed -n '$=' file.txt
```

> **Counting lines without `wc`:** `sed -n '$=' file.txt` is a handy alternative to `wc -l file.txt`. Unlike `wc -l`, it counts the number of newline-terminated lines without any trailing space or filename in the output — useful in scripts where you need a clean integer.

The Quit Commands: `q` and `Q`

The `q` command tells SED to stop processing after the current line. It prints the pattern space (unless `-n` is active) and exits. The `Q` command (GNU sed only) does the same but skips the final print — it exits silently.

```
sed 'q' at line 3 of a 100-line file: line 1: read → process → print → continue line 2: read → process →
print → continue line 3: read → process → print → q fires → EXIT lines 4–100: never read
```

Basic quit usage

```
# Print only the first 5 lines then stop (equivalent to head -5)
sed '5q' file.txt

# Print up to and including the first line matching a pattern, then stop
sed '/ERROR/q' logfile.txt

# Print the first line only (equivalent to head -1)
sed '1q' file.txt

# Extract just the HTTP status line from a curl response
curl -sI https://example.com | sed '1q'
```

Quit with an exit code

GNU sed allows an optional exit code after `q` and `Q`. This is useful in shell scripts where the exit status signals a result:

```
# Exit with code 1 if the pattern is NOT found (check for required config)
sed -n '/required_setting/q 0' config.txt; echo $?
# Exits 0 if "required_setting" is found, 1 otherwise (default)
```

The `Q` command — quit without printing

```
# Print lines 1 through 4, stop before line 5 (no print for line 5)
sed '5Q' file.txt
# With q: lines 1,2,3,4,5 are printed (q prints line 5 then quits)
# With Q: lines 1,2,3,4   are printed (Q quits before printing line 5)
```

> **The difference in practice:**
>
> - `Nq` — prints lines 1 through N, then exits (equivalent to `head -N`)
> - `NQ` — prints lines 1 through N-1, then exits (like `head -N` minus one)
>
> Use `q` when you want to include the triggering line in output; use `Q` when you want to stop just before it.

Performance: Why `q` and `Q` Matter on Large Files

SED reads files sequentially line by line. On a 500,000-line log file, if you only need the first match of a pattern, running `sed -n '/pattern/p'` reads all 500,000 lines even after finding the first match. Adding `q` stops as soon as the match is found:

```
# Slow on large files — reads the ENTIRE file even after finding the match
sed -n '/ERROR/p' huge.log | head -1

# Fast — stops reading as soon as the first match is found and printed
sed -n '/ERROR/ { p; q }' huge.log

# Even faster with Q — stops without the final print overhead
sed '/ERROR/ { p; Q }' huge.log
# (the p runs first, then Q exits — no default print attempted)
```

> **Always use `q` or `Q` when you only need the first match (or the first N lines) from a large file.** The difference in speed can be dramatic — finding a match on line 100 of a million-line file using `q` is 10,000× faster than processing all million lines.

Combining `d`, `p`, and `q`

The real power emerges when these commands work together. Here are patterns you will use repeatedly:

grep-like filtering with SED

```
# Print only matching lines — equivalent to: grep "pattern" file
sed -n '/pattern/p' file.txt

# Print non-matching lines — equivalent to: grep -v "pattern" file
sed '/pattern/d' file.txt

# Print matching lines with line numbers — equivalent to: grep -n "pattern" file
sed -n '/pattern/ { =; p }' file.txt
```

head and tail equivalents

```
# First N lines — equivalent to: head -N file
sed 'Nq' file.txt          # replace N with a number

# Last N lines — equivalent to: tail -N file
# (requires knowing total line count or using a different approach)
lines=$(sed -n '$=' file.txt)
sed -n "$((lines - N + 1)),\$p" file.txt

# Skip first N lines — equivalent to: tail -n +N+1 file
sed '1,Nd' file.txt         # replace N with a number
```

Extracting a specific section

```
# Print lines 50-60 then stop (efficient — stops at line 60)
sed -n '50,60p'; '60q' file.txt

# Better: combine the address and quit in one expression
sed -n '50{ :loop; p; n; 60q; b loop }' file.txt

# Simplest: just use the range address — SED is smart enough
sed -n '50,60p' file.txt
```

Viewing a section of a log file around an error

```
# Print the first ERROR line and stop
sed -n '/ERROR/ { p; q }' app.log

# Print everything up to the first ERROR (exclusive) then stop
sed '/ERROR/Q' app.log

# Print everything from the first ERROR to end of file
sed -n '/ERROR/,${p}' app.log
```

Removing duplicate consecutive lines

```
# Delete a line if it is identical to the previous one
# (uses hold space — covered fully in Chapter 6)
sed '$!N; /^\(.*\)\n\1$/!P; D' file.txt
# Brief explanation: N appends next line; if the two lines match, D deletes
# the first without printing; P prints the first if they differ
```

Practical Recipes

Extract email addresses from a file

```
# Print only lines that look like email addresses
sed -En '/[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}/p' contacts.txt
```

View only the active (uncommented) lines of a config

```
sed '/^[[:space:]]*#/d; /^[[:space:]]*$/d' /etc/ssh/sshd_config
```

Check if a pattern exists and exit appropriately

```
# Use in a shell script: exit 0 if pattern found, exit 1 if not
if sed -n '/required_key/q 0; $q 1' config.txt; then
  echo "Config OK"
else
  echo "ERROR: required_key missing from config"
fi
```

Print function signatures from a C or shell file

```
# Print lines that look like shell function declarations
sed -n '/^[a-zA-Z_][a-zA-Z0-9_]*[[:space:]]*()[[:space:]]*{/p' script.sh

# Print C function signatures (very simplified)
sed -n '/^[a-z].*([^;]*$/{/^#/d; p}' source.c
```

Fast search through a large compressed log

```
# Decompress on-the-fly and stop at first match
zcat archive.log.gz | sed -n '/CRITICAL/ { p; q }'
```

Strip a shebang line from a script

```
# Remove the first line if it starts with #!
sed '1{ /^#!/d }' script.sh
```

Command Interaction Summary

Command	Prints pattern space?	Continues script?	Reads next line?	POSIX?
`d`	No	No — aborts rest of script	Yes (next cycle)	Yes
`p`	Yes (immediately)	Yes — script continues	Yes (at end of cycle)	Yes
`=`	No — prints line number	Yes — script continues	Yes (at end of cycle)	Yes
`q`	Yes (unless `-n`)	No — exits SED	No — SED terminates	Yes
`Q`	No	No — exits SED	No — SED terminates	GNU only

Quick Reference — Chapter 4

DELETE

`d`	Delete pattern space; start next cycle immediately (no print)
`/pattern/d`	Delete lines matching *pattern*
`/pattern/!d`	Delete lines NOT matching *pattern* (keep only matches)
`n,md`	Delete lines *n* through *m*

PRINT

`p`	Print pattern space now (also prints at end of cycle unless `-n`)
`-n … p`	With `-n`: only print explicitly — turns SED into a filter
`=`	Print the current line number
`sed -n '$='`	Print total number of lines in file

QUIT

`q`	Print pattern space then exit (stop reading input)
`Q`	Exit immediately without printing (GNU sed)
`Nq`	Print first N lines then stop (like `head -N`)
`q N`	Quit with exit code N (GNU sed)
`/pat/ { p; q }`	Print first matching line then stop (efficient on large files)

What is coming next: Chapter 5 covers the `a` (append), `i` (insert), and `c` (change) commands — the tools for adding new lines to output and replacing entire lines or blocks. These are essential for tasks like inserting file headers, adding config directives, and replacing stale blocks of content.

CHAPTER 05

Append, Insert and Change

Chapter 5 — Append, Insert and Change

Adding and Replacing Lines

The substitute command (`s`) works within a line — it replaces matched text inside the pattern space. But sometimes you need to do something more structural: add a new line after a match, insert a line before it, or replace the entire line with something different. That is what `a` , `i` , and `c` are for.

Command	Full name	What it does	When it outputs
`a`	Append	Adds one or more lines *after* the addressed line	After the current line is printed at end of cycle
`i`	Insert	Adds one or more lines *before* the addressed line	Immediately, before the pattern space is printed
`c`	Change	Replaces the addressed line(s) with new text	In place of the addressed line; suppresses normal print

A crucial difference from `s` : the text supplied to `a` , `i` , and `c` is **literal** — it is not treated as a regular expression, and special characters like `&` and `\1` have no meaning in it. What you type is exactly what gets output.

The Append Command: `a`

The `a` command queues text to be printed *after* the current line. The appended text does not become part of the pattern space — it is output directly after the current cycle's default print, without passing through any further SED commands.

Syntax — two forms

GNU sed accepts a concise single-line form:

```
sed 'a\text to append' file.txt
# Or without the backslash (GNU sed extension):
sed 'a text to append' file.txt
```

The POSIX-portable form uses a backslash-newline after the command letter, with the text on the following line:

```
sed 'a\
text to append' file.txt
```

In a script file (`-f`) the portable form is always safe and clearest:

```
# append.sed
a\
text to append
```

Appending after every line

```
# Add a blank line after every line (double-space a file)
sed 'a\\' file.txt

# Add a separator after every line
sed 'a\---' file.txt
```

Appending after a specific line

```
# Add a line after line 3
sed '3a\inserted after line 3' file.txt

# Add a line after the last line (append to end of file)
sed '$a\# End of configuration' config.txt
```

Appending after a pattern match

```
# Add a blank line after every section heading (lines starting with [)
sed '/^\[/a\\' config.ini
```

```
BEFORE
[database] host=localhost port=5432 [cache] ttl=300
```

```
AFTER SED '/^\[/A\\'
[database]   host=localhost port=5432 [cache]
ttl=300
```

```
# Add a new directive after an existing one in a config file
sed '/^ServerName/a\ServerAlias www.example.com' httpd.conf

# Add a comment explaining the next line
sed '/^PermitRootLogin/i\# Disable root login for security' sshd_config
# (note: this uses i not a — see below)
```

The Insert Command: `i`

The `i` command works exactly like `a` but places text *before* the addressed line rather than after it. The inserted text is output immediately, before SED prints the pattern space at end of cycle.

Syntax

```
# GNU sed single-line form
sed 'i\text to insert' file.txt
sed 'i text to insert' file.txt

# POSIX portable form
sed 'i\
text to insert' file.txt
```

Inserting before a specific line

```
# Insert a header before line 1 (prepend to file)
sed '1i\# Generated by deploy.sh — do not edit manually' output.conf

# Insert a blank line before every line starting with "Chapter"
sed '/^Chapter/i\\' book.txt
```

```
BEFORE
Introduction text. Chapter 1: Basics Content here.
Chapter 2: Advanced More content.
```

```
AFTER SED '/^CHAPTER/I\\'
Introduction text.
 Chapter 1: Basics Content here.
 Chapter 2: Advanced More content.
```

Inserting before a pattern match

```
# Add a warning comment before any line that sets a dangerous option
sed '/PermitRootLogin yes/i\# WARNING: root login enabled — review before production' sshd_config

# Insert a CSS rule before an existing one
sed '/\.header {/i\.header-wrapper { width: 100%; }' styles.css

# Add a shebang line to a script that is missing one
sed '1i\#!/usr/bin/env bash' script.sh
```

The Change Command: `c`

The `c` command replaces the entire addressed line — or the entire addressed range — with new text. It suppresses the normal printing of the pattern space and outputs the replacement text instead.

This is fundamentally different from `s`: while `s` modifies text *within* a line, `c` throws the line away entirely and puts something new in its place.

Syntax

```
# GNU sed single-line form
sed '/pattern/c\replacement line' file.txt

# POSIX portable form
sed '/pattern/c\
replacement line' file.txt
```

Replacing a specific line by number

```
# Replace line 1 entirely
sed '1c\# New header line' file.txt

# Replace the last line
sed '$c\# End of file' file.txt
```

Replacing matched lines

```
# Replace any line containing "TODO" with a placeholder
sed '/TODO/c\# [REMOVED: TODO item]' notes.txt

# Completely replace a config directive with a known-good value
sed '/^MaxConnections/c\MaxConnections 100' server.conf
```

```
BEFORE
ServerName example.com MaxConnections 9999 Timeout 300 MaxConnections 0
```

```
AFTER SED '/^MAXCONNECTIONS/C\MAXCONNECTIONS 100'
ServerName example.com MaxConnections 100 Timeout 300 MaxConnections 100
```

> **Every matching line is replaced independently.** If the pattern matches five lines, each is replaced by the same `c` text, so the replacement text appears five times. This is different from a range address — see below.

Replacing a range with `c`

When `c` is applied to a *range* address, the behaviour changes: instead of replacing each line in the range individually, the **entire range is replaced with the text just once** — output when the last line of the range is processed.

```
# Replace lines 3 through 7 with a single placeholder line
sed '3,7c\[section removed]' file.txt
```

```
BEFORE
line 1 line 2 line 3 line 4 line 5 line 6
```

```
AFTER SED '3,5C\[REMOVED]'
line 1 line 2 [removed] line 6
```

```
# Replace an entire config block with a new version
sed '/^# BEGIN SSL/,/^# END SSL/c\# SSL configured via include — see ssl.conf' httpd.conf
```

Range + pattern address quirk with `c`: When using a pattern range (`/start/,/end/`) with `c`, the single-output behaviour applies only for line-number ranges. With pattern ranges, each line in the matching range is still processed individually by `c` (outputting the replacement text for each line). Use `d` + `a` together if you need a true block replacement with a pattern range.

Multi-line Text Blocks

All three commands — `a`, `i`, and `c` — can output multiple lines. In the multi-line form, each line except the last ends with a backslash:

```
# Append a two-line block after line 1
sed '1a\
# ======================\
# Auto-generated config\
# Do not edit manually' config.txt

# Insert a multi-line licence header before line 1
sed '1i\
# Copyright 2025 Example Corp\
# Licensed under the MIT Licence\
#' script.sh

# Replace a line with a multi-line block
sed '/^PLACEHOLDER/c\
# Line one of replacement\
# Line two of replacement\
# Line three of replacement' template.txt
```

For large multi-line insertions in scripts, use a here-doc approach instead. Feeding a multi-line SED script via `-f` is far more readable than a shell one-liner when you are inserting five or more lines.

Comparing `a`, `i`, `c` and `s`

Command	Operates on	Text is literal?	Pattern space changed?	Output position
`s`	Text within the current line	No — replacement has special chars (`&` , `\1`)	Yes — match is replaced in-place	At end of cycle (normal print)
`a`	A new line added after the current one	Yes — completely literal	No	After the current line's normal print
`i`	A new line added before the current one	Yes — completely literal	No	Before the current line's normal print
`c`	The entire current line (or range)	Yes — completely literal	Yes — pattern space deleted	Replaces the normal print

Practical Recipes

Add a file header and footer

```
# Prepend a header and append a footer to any file
sed -e '1i\# AUTO-GENERATED — DO NOT EDIT' \
    -e '$a\# END OF FILE' output.conf
```

Insert a new config directive after an existing one

```
# Add "AllowOverride All" after every "DocumentRoot" line
sed '/DocumentRoot/a\    AllowOverride All' httpd.conf

# Add "Restart=on-failure" after "Type=simple" in a systemd unit
sed '/^Type=simple/a\Restart=on-failure' myservice.service
```

Replace a placeholder in a template

```
# template.conf contains the line: ##DB_CONFIG_PLACEHOLDER##
# Replace it with the real database block
sed '/^##DB_CONFIG_PLACEHOLDER##/c\
db_host=db.internal\
db_port=5432\
db_name=production\
db_user=app' template.conf
```

Add blank lines around every heading in a Markdown file

```
# Insert blank line before and append blank line after every ## heading
sed '/^## / {
  i\\
  a\\
}' README.md
```

Wrap each line of a file in XML tags

```
# Transform each line into <item>line content</item>
# s// handles the wrapping inline — but a and i can add surrounding tags
sed 's/.*\/<item>&<\/item>/' items.txt
# Or using i and a for a different structure:
sed -e '1i\<items>' \
    -e 's/.*\/  <item>&<\/item>/' \
    -e '$a\<\/items>' items.txt
```

Safely replace a known-bad config value

```
# Use c rather than s when the entire line should be replaced
# — avoids accidentally matching part of a value
sed '/^net.ipv4.ip_forward/c\net.ipv4.ip_forward = 1' /etc/sysctl.conf
```

Add a redirect rule to an nginx config

```
# Insert a return 301 after the server_name directive
sed '/server_name old-domain.com/a\    return 301 https://new-domain.com$request_uri;' nginx.conf
```

Remove a block and replace with a comment

```
# Between BEGIN LEGACY and END LEGACY, replace the whole block
sed '/# BEGIN LEGACY/,/# END LEGACY/ {
  /# BEGIN LEGACY/c\# LEGACY CODE REMOVED — see migration guide
  /# BEGIN LEGACY/!d
}' script.sh
# Explanation:
#  On the BEGIN line: c replaces it with the comment
#  On all other lines in the range: d deletes them
```

Portability Notes

Feature	GNU sed (Linux)	BSD sed (macOS)
`a text` (no backslash)	Works	Requires `a\` with newline
`i text` (no backslash)	Works	Requires `i\` with newline
`c text` (no backslash)	Works	Requires `c\` with newline
Range address with `c`	Outputs replacement once for entire range	Outputs replacement once per line in range
Multi-line text with `\`	Works	Works

Most portable style: Always use the backslash-newline form in shell scripts or `-f` files that need to run on both Linux and macOS:

```
sed '/pattern/a\
text to append' file.txt
```

This works on GNU sed, BSD sed, and all POSIX-compliant implementations without modification.

Quick Reference — Chapter 5

APPEND — A

`a\text`	Append *text* after every line
`Na\text`	Append *text* after line *N*
`/pat/a\text`	Append *text* after every line matching *pat*
`$a\text`	Append *text* after the last line (add to end of file)

INSERT — I

`i\text`	Insert *text* before every line
`1i\text`	Insert *text* before line 1 (prepend to file)
`/pat/i\text`	Insert *text* before every line matching *pat*

CHANGE — C

`/pat/c\text`	Replace every line matching *pat* with *text*
`n,mc\text`	Replace lines *n* through *m* with a single *text* line
`Nc\text`	Replace line *N* entirely with *text*

```
MULTI-LINE TEXT (ALL THREE COMMANDS)

a\line one\                          Each line except the last ends with \ to continue
         line two
```

What is coming next: Chapter 6 introduces the hold space — SED's second buffer that persists between processing cycles. It is the key to solving problems that require memory of a previous line, such as reversing a file, deduplicating, or processing header–body–footer structures. This is where SED moves from a line-by-line tool to something genuinely powerful.

CHAPTER 06

The Hold Space

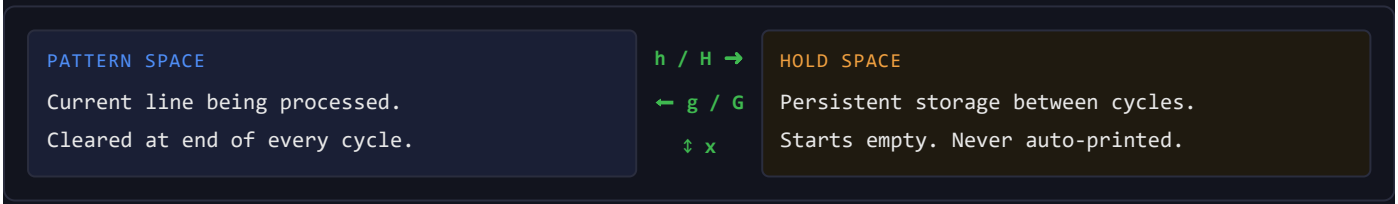
Chapter 6 — The Hold Space

Two Buffers, Not One

Until now, every SED command has worked with a single piece of memory — the **pattern space** — which holds the current line and is cleared at the end of every cycle. This works perfectly for transformations that only need to look at one line at a time.

But some problems require memory that survives from one line to the next. How do you reverse the lines of a file? How do you print a line only if the *next* line matches something? How do you collect a group of lines and reassemble them? You cannot solve these with only the pattern space, because it is wiped clean every cycle.

SED's answer is the **hold space** — a second buffer that persists across cycles. It starts empty and keeps whatever you put in it until you change it or SED finishes. The hold space is never automatically cleared, never automatically printed, and never automatically written. You control it entirely with five commands.



The Five Hold Space Commands

Command	Full name	What it does
`h`	hold (copy)	**Copy** pattern space → hold space (overwrites hold space)
`H`	Hold (append)	**Append** a newline + pattern space → hold space
`g`	get (copy)	**Copy** hold space → pattern space (overwrites pattern space)
`G`	Get (append)	**Append** a newline + hold space → pattern space
`x`	exchange	**Swap** the contents of pattern space and hold space

The capitalisation rule is consistent and worth memorising: **lowercase copies** (overwriting the destination), **uppercase appends** (adding to the destination with a separating newline). The direction is always: `h` / `H` go *to* the hold space; `g` / `G` come *from* it; `x` swaps both ways simultaneously.

Understanding Each Command

`h` — copy pattern space to hold space

The pattern space is copied into the hold space, *replacing* whatever was there. Think of it as saving the current line for later.

```
# Save line 3 for later use
sed '3h' file.txt
# (nothing changes in output — h just saves, it doesn't print anything)
```

`H` — append pattern space to hold space

A newline followed by the pattern space is appended to the hold space. The hold space grows line by line. This is how you accumulate multiple lines into a single buffer.

```
# Accumulate all lines into the hold space, print everything at end
sed -n 'H; ${g; p}' file.txt
# H appends each line; $ triggers g (copy hold→pattern) and p (print)
# Result: the entire file in the pattern space on the last line
# Note: the hold space starts empty so the first H adds a leading newline
```

> **H always prepends a newline before appending.** Because the hold space starts empty, the very first `H` in a run produces a leading newline (empty string + newline + first line). This is usually harmless but watch for it in scripts that process the accumulated content. Use `1{h;d}; H` instead of plain `H` to avoid the leading newline (see recipes below).

`g` — copy hold space to pattern space

The hold space is copied into the pattern space, *replacing* the current line. The old pattern space content is lost. This is how you retrieve what you saved earlier and do something with it.

```
# On the last line, replace the pattern space with the hold space contents
sed -n '${ g; p }' file.txt
```

`G` — append hold space to pattern space

A newline followed by the hold space is appended to the pattern space. The current line is preserved and the held content follows it. Useful for inserting held content after every line, or for building combined lines.

```
# Append the hold space (whatever was saved) after every line
sed '1h; G' file.txt
# h saves line 1 into hold; G appends it after every subsequent line
```

`x` — exchange pattern space and hold space

The contents of the pattern space and hold space are swapped in a single atomic operation. Neither is lost. This is the most powerful of the five commands because it allows you to work with the saved content while keeping a reference to the current line.

```
# Print the previous line each time a match is found
sed -n '/ERROR/ { x; p; x }' logfile.txt
# x brings the previous line into the pattern space
# p prints it
# x swaps back so the cycle continues normally
```

Step-by-Step Traces

The hold space is much easier to understand by watching it change cycle by cycle. Here are two detailed traces.

Trace 1: Printing the previous line on a match

Problem: print the line immediately *before* every line containing "ERROR".

Script: `sed -n '/ERROR/{x;p;x}; x'`

```
Input: "line A" / "line B" / "ERROR here" / "line D"

Cycle 1 — read "line A" /ERROR/ → no match, block skipped x → PS="" (was hold), HS="line A" -n → no default print Cycle 2 — read "line B" /ERROR/ → no match, block skipped x → PS="line A", HS="line B" -n → no default print Cycle 3 — read "ERROR here" /ERROR/ → matches! enter block x → PS="line B", HS="ERROR here" p → prints "line B" x → PS="ERROR here", HS="line B" x → PS="line B", HS="ERROR here" -n → no default print Cycle 4 — read "line D" /ERROR/ → no match x → PS="ERROR here", HS="line D" Final output: "line B"
```

Trace 2: Reversing two lines

Problem: swap every pair of lines (line 2 before line 1, line 4 before line 3, etc.).

Script: `sed -n 'h;n;p;g;p'`

```
Input: "first" / "second" / "third" / "fourth"

Cycle 1 — read "first" h → HS="first" n → read next line; PS="second" (advances to next line) p → prints "second" g → PS="first" (from hold) p → prints "first" Cycle 2 — read "third" (n consumed "second", so next unread line is "third") h → HS="third" n → PS="fourth" p → prints "fourth" g → PS="third" p → prints "third" Final output: second / first / fourth / third
```

Classic Hold Space Recipes

Reverse the lines of a file (tac equivalent)

This is the most famous hold space pattern. It accumulates every line into the hold space in reverse order, then prints it all at the end:

```
# Reverse all lines — equivalent to: tac file.txt
sed -n '1!G; h; $p' file.txt

# Broken down:
#   1!G  — on every line except line 1: append hold→pattern
#          (so pattern = current line + newline + everything before it)
#   h    — copy that combined content back to hold space
#   $p   — on the last line: print the accumulated hold content
```

```
Input: "a" / "b" / "c"

Cycle 1 — PS="a" 1!G skipped (IS line 1) h → HS="a" Cycle 2 — PS="b" G → PS="b\na" h → HS="b\na" Cycle 3 —
PS="c" G → PS="c\nb\na" h → HS="c\nb\na" $p → prints c / b / a
```

Print the line before and after a match (context lines)

```
# Print the line BEFORE a match (using x to keep the previous line in hold)
sed -n '/ERROR/ { x; p }; x' logfile.txt

# Print the line AFTER a match (use n to advance and then print)
sed -n '/ERROR/ { n; p }' logfile.txt

# Print the matching line AND the line after it
sed -n '/ERROR/ { p; n; p }' logfile.txt
```

Accumulate lines into a block, then process the block

```
# Collect all lines of a paragraph (separated by blank lines) and print as one
sed -n '/^$/ { g; s/\n/ /g; p; d }; H' paragraphs.txt
# H accumulates lines into hold space
# On blank line: g retrieves, joins newlines with spaces, prints, resets
```

Remove duplicate consecutive lines (uniq equivalent)

```
# Delete a line if it is identical to the previous one
sed '$!N; /^\(.*\)\n\1$/!P; D' file.txt

# Broken down:
#   $!N             — on all but last line: append next line to pattern space
#                     pattern space now = "line1\nline2"
#   /^\(.*\)\n\1$/  — does pattern space match "X\nX" (two identical lines)?
#   !P              — if NOT identical: print first line only (P = multiline print)
#   D               — delete first line, restart cycle with second line in pattern space
```

Swap every pair of lines

```
# Print line 2 before line 1, line 4 before line 3, etc.
sed -n 'h; n; p; g; p' file.txt
# h saves current line; n reads the next line and prints it first;
# g retrieves saved line; p prints it second
```

Move a line to a different position

```
# Take line 1 (a header), delete it there, and append it after line 5
sed -n '1{ h; d }; 5{ p; g; p; d }; p' file.txt
# Line 1: save to hold, delete (skip print)
# Line 5: print it, then retrieve and print line 1 after it
# All other lines: print normally
```

Double-space a file (add blank line after every line)

```
# Using G with an empty hold space to insert a blank line after each line
sed 'G' file.txt
# The hold space starts empty, so G appends "\n" + "" = just a newline
# Result: every line is followed by a blank line
```

Delete a line and the one immediately following it

```
# Delete any line matching "REMOVE" and the line that follows it
sed '/REMOVE/ { N; d }' file.txt
# N appends the next line into the pattern space
# d deletes both lines together
```

Print only unique lines (remove ALL duplicates, not just consecutive)

```
# Collect all lines in hold space, then sort and uniq — but that needs a loop
# For a pure-SED approach (order-preserving, marks duplicates):
sed -n 'G; /^\(.*\)\n.*\1/!{ s/\n.*//; p; h }; s/.*\n//; H' file.txt
# This is complex — in practice, use: awk '!seen[$0]++'
```

The `n` and `N` Commands — Reading Ahead

The hold space commands are often used alongside two related commands that control how SED reads input. They are not hold space commands themselves, but they are closely related in practice and are essential for many multi-line patterns.

Command	What it does	Effect
`n`	Next — read the next line	Prints the current pattern space (unless `-n`), then reads the next input line into the pattern space, replacing the current one. The cycle continues from this point.
`N`	Next append — append the next line	Appends a newline + the next input line to the pattern space (does *not* print). The pattern space now contains two lines. Covered more fully in Chapter 7.

```
# n — skip the line after every blank line
sed '/^$/ { n; d }' file.txt
# When a blank line is found: n reads the next line into the pattern space,
# then d deletes it — effectively removing the line after every blank

# n — print every other line (even lines)
sed -n 'n; p' file.txt
# Cycle reads line 1 (odd), n reads line 2 (even) into pattern space, p prints it

# n — process pairs: substitute only on the second of each pair
sed '/^LABEL/ { n; s/value=.*/value=NEW/ }' config.txt
```

Avoiding Common Hold Space Mistakes

The leading newline from H

```
# PROBLEM: H on line 1 adds a blank line at the start
printf 'a\nb\nc\n' | sed -n 'H; ${ g; p }'
# Output has a leading blank line because hold starts empty

# FIX: use h on line 1 to initialise without a leading newline
printf 'a\nb\nc\n' | sed -n '1h; 1!H; ${ g; p }'
a
b
c
```

Forgetting that g wipes the pattern space

```
# PROBLEM: after g, the current line is gone
sed '/marker/ { h; g; s/marker/FOUND/ }' file.txt
# h saves the line, then g replaces the pattern space with the hold space
# — but hold space IS the same content, so this is a no-op here
# More dangerous: using g when the hold space has different content

# Use x to swap (preserving both) rather than g when you need both values
```

x on the first line exchanges with empty hold space

```
# The hold space starts empty — the first x puts that empty string in pattern space
printf 'a\nb\n' | sed -n 'x; /./p'
a
# Cycle 1: x swaps "a" to hold, "" to pattern — /./p doesn't print (pattern is empty)
# Cycle 2: x swaps "b" to hold, "a" to pattern — /./p prints "a"
# "b" is in hold space at end — never printed
# This is the "print previous line" pattern — intentional here, but easy to be caught out by
```

Practical Recipes

Extract a named section from an INI/TOML file

```
# Print only the [database] section (from header to next header or EOF)
sed -n '/^\[database\]/,/^\[/{/^\[database\]/!{ /^\[/d }; p}' config.ini

# Simpler approach using hold to stop at next section:
sed -n '/^\[database\]/{ :loop; p; n; /^\[/q; b loop }' config.ini
```

Join continuation lines (lines ending with \)

```
# Collapse lines ending in \ onto the next line (like make/shell line continuation)
sed -n ':join; /\\$/ { N; s/\\\n//; b join }; p' Makefile
```

Swap the first and last lines of a file

```
sed -n '1{ h; d }; 2,${p}; ${ g; p }' file.txt
# Line 1: save to hold, delete
# Lines 2 to second-to-last: print normally
# Last line: print it (already happened via 2,$p), then retrieve and print line 1
```

Print lines between two patterns, excluding the markers

```
sed -n '/^START$/ { n; /^END$/q; :l; p; n; /^END$/q; b l }' file.txt
```

Annotate each line with the previous line as a comment

```
sed 'G; s/\n/\n# Previous: /' file.txt
# G appends the hold space (previous line) after the current line
# s replaces the joining newline with a newline + "# Previous: " prefix
```

Quick Reference — Chapter 6

PATTERN SPACE → HOLD SPACE

`h`	Copy pattern space to hold space (overwrites hold)
`H`	Append newline + pattern space to hold space

HOLD SPACE → PATTERN SPACE

`g`	Copy hold space to pattern space (overwrites pattern space)
`G`	Append newline + hold space to pattern space

EXCHANGE

`x`	Swap pattern space and hold space simultaneously

RELATED — READING AHEAD

`n`	Print current pattern space, then read next line into it
`N`	Append next line to pattern space (no print) — see Chapter 7

KEY PATTERNS

`1!G; h; $p`	Reverse all lines (tac equivalent)
`G`	Double-space a file (hold starts empty → blank line after each)
`1h; 1!H; ${g;p}`	Accumulate all lines, print as single block at end
`/pat/{x;p};x`	Print the line before every match

What is coming next: Chapter 7 covers multi-line commands in depth — `N`, `P`, and `D`. These three commands let SED work across line boundaries: joining lines together, printing only part of a multi-line pattern space, and looping back to the start of the script with modified content. They are the foundation of SED's ability to process records that span multiple lines.

CHAPTER 07

Multi-line Commands

Chapter 7 — Multi-line Commands

Crossing the Line Boundary

Every technique covered so far has processed one line at a time. Each cycle reads a single line into the pattern space, the commands run, the line is printed, and the pattern space is cleared. The line boundary is the fundamental unit of SED processing.

But real text does not always break conveniently at line boundaries. An XML tag might open on one line and close on another. A function call might be split across two lines. An error message might span a header line and a detail line. To process these structures, SED provides three commands that deliberately cross the line boundary — `N`, `P`, and `D`. Together they form a powerful trio for multi-line work.

N

NEXT APPEND

Read the next input line and **append** it to the pattern space, separated by a newline. The pattern space now holds two (or more) lines.

P

PRINT FIRST

Print only the **first line** of the pattern space — everything up to the first embedded newline. Leaves the pattern space unchanged.

D

DELETE FIRST

Delete only the **first line** of the pattern space and **restart the script** from the beginning with the remainder — without reading a new line.

N — Appending the Next Line

The lowercase `n` command (Chapter 4) replaces the pattern space with the next line. The uppercase `N` is different: it *appends* the next line to whatever is already in the pattern space, inserting a literal newline character between them.

```
# After N fires, the pattern space contains TWO lines joined by \n
printf 'line one\nline two\nline three\n' | sed 'N; s/\n/ /'
line one line two
line three
# N joins line 1 and line 2 into the pattern space
# s/\n/ / replaces the embedded newline with a space → "line one line two"
# Line three is processed alone (no pair to join with)
```

```
PATTERN SPACE AFTER N FIRES ON "LINE ONE"
line one\nline two ↑ original line ↑ embedded newline ↑ appended line
```

Key behaviours of N

- The pattern space now contains an embedded `\n` — you can match across it with `\n` in a regex
- The line counter is advanced (the next line has been consumed)
- If there is no next line (N fires on the last line), GNU sed prints the pattern space and exits — it does not error
- You can call N repeatedly in a loop to accumulate more than two lines

> **N on the last line:** In GNU sed, if `N` tries to read past the last line, SED prints the pattern space and exits. In POSIX/BSD sed, this terminates the script without printing. Guard against this with `$!N` (apply N on all lines except the last) when portability matters.

Matching across a line boundary

```
# Find "foo" on one line followed by "bar" on the next, join them
sed '/foo/ { N; /bar/ s/foo\nbar/foobar/ }' file.txt

# Remove a line break between any line ending with a comma and the next
sed ':a; /,$/ { N; s/,\n/,/; ta }' data.txt
# :a is a label; ta branches back to :a if a substitution succeeded
# This loops, consuming continuation lines until no trailing comma remains

# Delete a pattern that spans two lines
sed '/^BEGIN$/ { N; /^BEGIN$\nEND$/ d }' file.txt
```

Substituting across line boundaries

```
# Replace a newline between two specific patterns with a space
sed '/^Subject:/ { N; s/\n[[:space:]]/ / }' email.txt
# Joins folded email headers (RFC 2822 header continuation)

# Collapse a multi-line function call onto one line (lines ending with ,)
sed -E ':a; N; s/,\n[[:space:]]*/,/; ta' source.js
```

P — Printing the First Line

Once the pattern space holds multiple lines (thanks to `N` or hold space operations), you often need to output only the first line — not the entire accumulated content. That is what `P` does. It prints everything up to the first embedded `\n`.

```
# P vs p on a two-line pattern space
printf 'a\nb\nc\n' | sed -n 'N; P'
a
c
# N joins a+b into pattern space; P prints only "a"
# cycle ends (no default print due to -n)
# Next cycle reads "c" alone (b was consumed by N); P on single line prints "c"
```

Compare the three print behaviours side by side:

Command	What is printed	Pattern space after
`p`	The *entire* pattern space (all lines)	Unchanged
`P`	Only the *first line* of the pattern space	Unchanged
`d`	Nothing — suppresses output	Cleared; next cycle starts
`D`	Nothing — suppresses first line	First line removed; script restarts with remainder

D — Deleting the First Line and Restarting

`D` is the most complex of the three. It does two things atomically:

1. Deletes the first line of the pattern space (everything up to and including the first `\n`)
2. Restarts the script from the beginning — *without reading a new line from input*

This restart behaviour is what makes `D` powerful. After deleting the first line, the remainder of the pattern space (the second line and beyond) is still there, and the full script runs again on that remaining content. `D` creates an implicit loop.

```
D ON PATTERN SPACE "LINE ONE\NLINE TWO"
line one\n ← deleted by D line two ← remains; script restarts here without reading input
```

```
# Practical effect: D is like d but keeps the second line "in play"
printf 'a\nb\nc\n' | sed 'N; P; D'
a
b
c
# This is a sliding-window loop — prints each line once:
# Cycle 1: N → PS="a\nb"; P → prints "a"; D → PS="b", restart script
# Script restart: N → PS="b\nc"; P → prints "b"; D → PS="c", restart
# Script restart: N on last line → PS="c" (no more input); P → prints "c"; D → empty, done
```

The N-P-D Loop — A Sliding Window

The combination `N; P; D` is a fundamental SED idiom. It creates a **sliding window** of two lines: at any moment, the pattern space holds the current line and the next. After each step, the window slides forward by one. This gives every SED command the ability to look at two consecutive lines simultaneously.

```
sed 'N; /foo.*\nbar/d; P; D' — delete "foo" line when followed by "bar"
Input: "one" / "foo here" / "bar here" / "four"
Window 1: PS = "one\nfoo here" /foo.*\nbar/ → no match (foo is on line 2, not line 1) P → print "one" D →
PS="foo here", restart script Window 2: N appends → PS = "foo here\nbar here" /foo.*\nbar/ → matches! d → both
lines deleted, next cycle Window 3: PS = "four" (last line, N exits) print "four" Final output: one / four
```

Step-by-Step Traces

Trace: Remove duplicate consecutive lines

Script: `sed '$!N; /^\(.*\)\n\1$/!P; D'`

Input: "apple" / "apple" / "banana" / "banana" / "cherry"

Cycle 1 — PS="apple" $!N → PS="apple\napple" /^\(.*\)\n\1$/ → "apple\napple" MATCHES (both lines identical) !P → negated, so P does NOT fire (duplicate found — suppress) D → PS="apple", restart Restart — PS="apple" (the second apple) $!N → PS="apple\nbanana" /^\(.*\)\n\1$/ → no match (different lines) !P → fires → prints "apple" D → PS="banana", restart Restart — PS="banana" $!N → PS="banana\nbanana" → duplicate found → P suppressed → D → PS="banana" Restart — PS="banana" (second banana) $!N → PS="banana\ncherry" → no match → prints "banana" → D → PS="cherry" Restart — PS="cherry" (last line) $!N → $ matches, N skipped no match → prints "cherry" → D clears → done Final output: apple / banana / cherry

Practical Recipes

Join lines that end with a backslash (shell/Makefile continuation)

```
# Collapse continuation lines ending in \ into one logical line
sed ':a; /\\$/ { N; s/\\\n//; ta }' Makefile
# :a   — label for looping
# /\\$/ — if current (accumulated) line ends with a backslash
# N    — append the next line
# s/\\\n// — remove the backslash-newline join point
# ta   — if substitution succeeded, loop back to :a
```

Join folded email/HTTP headers

```
# RFC 2822: header continuation lines start with a space or tab
sed ':a; N; s/\n[[:space:]]/ /; ta; P; D' headers.txt
# Accumulate lines, join continuation whitespace, slide the window forward
```

Remove blank lines between paragraphs (collapse multiple blanks into one)

```
# Replace two or more consecutive blank lines with a single blank line
sed '/^$/ { N; /^\n$/ d }' file.txt

# More robust version using the N-P-D loop:
sed ':a; /^$/ { $!N; /^\n/{ s/^\n//; ta } }' file.txt
```

Delete a line and the line immediately after it

```
# Delete any line matching "REMOVE" together with the line that follows
sed '/REMOVE/ { N; d }' file.txt
```

Delete a line and the line immediately before it

```
# Using the sliding window: delete the window when the second line matches
sed -n '$!N; /\nREMOVE$/!P; D' file.txt
# P only fires when the second line does NOT match — suppressing both lines when it does
```

Wrap long lines at word boundaries

```
# Split any line longer than 72 characters at the last space before position 72
sed -E ':a; s/^(.{72}[^ ]*) (.)/\1\n\2/; ta' longlines.txt
# Matches 72+ chars followed by a space and next char, inserts a newline at the break
# ta loops until no more splits needed on this line
```

Convert Unix line endings back to Windows (add \r before \n)

```
# GNU sed: insert carriage return before every newline
sed 's/$/\r/' unix.txt > windows.txt
```

Find and remove a pattern that spans exactly two lines

```
# Delete any two-line block where line 1 = "START" and line 2 = "END"
sed '/^START$/ { N; /^START\nEND$/ d }' file.txt
```

Substitute across a line boundary — replace split tag

```
# An opening XML tag is sometimes split across two lines in generated output
# <longTag
#   attr="value">
# Join and normalise it:
sed '/<[A-Za-z]/ { :a; />/!{ N; ta }; s/\n[[:space:]]*/  /g }' file.xml
```

Print paragraphs containing a keyword

```
# A paragraph is a block of non-blank lines separated by blank lines
# Accumulate each paragraph, print if it contains "keyword"
sed -n '/^$/ { x; /keyword/p; d }
        H' file.txt
# H accumulates lines into hold space
# On blank line: x swaps hold to pattern, check for keyword, print or discard
```

Split a CSV line at every comma onto separate lines

```
# Turn "a,b,c,d" into four separate lines
sed 's/,/\n/g' data.csv

# More controlled: only split the first field off
sed 's/,/\n/' data.csv
```

n vs N — The Critical Difference

It is easy to confuse lowercase `n` and uppercase `N`. The distinction is essential:

	`n` (lowercase)	`N` (uppercase)
What it does	Prints pattern space (unless `-n`), then *replaces* it with the next line	*Appends* the next line to the pattern space (with an embedded `\n`)
Pattern space after	Contains only the new line	Contains the old line + `\n` + new line
Old content	Printed (or suppressed with `-n`) — gone from pattern space	Still in pattern space — not lost
Use when you want to	Move to the next line and continue processing	Combine two (or more) lines for cross-boundary matching

```
# n — move forward: prints "a", then processes "b"
printf 'a\nb\n' | sed 'n; s/b/B/'
a
B

# N — join: pattern space = "a\nb", then substitute across the boundary
printf 'a\nb\n' | sed 'N; s/a\nb/a+b/'
a+b
```

P vs p and D vs d

Similarly, it is worth keeping the uppercase/lowercase pairs clearly separate in your mind:

	Lowercase — acts on whole pattern space	Uppercase — acts on first line only
Print	`p` — prints entire pattern space	`P` — prints up to first `\n`
Delete	`d` — deletes entire pattern space, starts new cycle	`D` — deletes first line, *restarts script* with remainder

> **Memory aid:** Uppercase commands (`N` , `P` , `D`) all operate in a way that keeps processing going within the same logical context — they are the "multi-line aware" versions. Lowercase commands (`n` , `p` , `d`) are simpler, single-line operations that complete cleanly.

Common Mistakes

N on the last line terminating unexpectedly

```
# If N fires on the last line and there is no next line to read:
# GNU sed: prints pattern space and exits normally
# BSD/POSIX sed: exits immediately without printing

# Safe pattern — skip N on the last line:
sed '$!N; s/\n/ /' file.txt
# $! = "not the last line" — N only fires when there is a next line to read
```

Forgetting that D restarts the script

```
# MISTAKE: assuming D just removes the first line and continues normally
sed 'N; D; s/foo/bar/' file.txt
# s/foo/bar/ never runs! D causes the script to restart from the top
# The substitution is unreachable
```

Infinite loop with D and no exit condition

```
# DANGER: D restarts the script — if the restarted script fires N again
# immediately, it will grow the pattern space forever until input is exhausted
# Always ensure your D-based loops have a termination condition
# Good pattern: condition on content before D, not just N; ... D blindly
```

Quick Reference — Chapter 7

THE MULTI-LINE TRIO

`N`	Append next input line to pattern space (separated by `\n`)
`P`	Print first line of pattern space (up to first `\n`)
`D`	Delete first line of pattern space; restart script without reading new input

KEY IDIOMS

`$!N; P; D`	Sliding two-line window — see both current and next line at once
`$!N; /^\(.*\)\n\1$/!P; D`	Remove duplicate consecutive lines (uniq equivalent)
`N; s/\n/ /`	Join every pair of lines with a space
`/pat/ { N; /pat\nnext/d }`	Delete a line and the one after it when both match
`:a; /\\$/ { N; s/\\\n//; ta }`	Join backslash-continuation lines

N VS N / P VS P / D VS D

`n`	Print + replace pattern space with next line
`N`	Append next line to pattern space (keeps both)
`p`	Print entire pattern space
`P`	Print first line of pattern space only
`d`	Delete pattern space; start new cycle
`D`	Delete first line; restart script with remainder

What is coming next: Chapter 8 covers branching and labels — `b`, `t`, `T`, and `:label`. These are the control flow tools that turn SED scripts from a linear sequence of commands into proper loops and conditional branches. The `ta` (branch-if-substitution-succeeded) pattern you have already seen in this chapter is one of the most important — Chapter 8 explains exactly how it works and shows the full range of what branching can do.

CHAPTER 08

Branching and Labels

Chapter 8 — Branching and Labels

Control Flow in SED

SED scripts normally execute top to bottom — command 1, command 2, command 3, and so on through to the end of the script for each line of input. Branching breaks that linearity. It lets you jump to a named point in the script (a **label**), skip sections, or loop back to repeat commands. This turns SED from a simple transformation pipeline into a genuine programming language — limited, but Turing-complete.

There are three branch commands and one label declaration:

`:label`

LABEL DECLARATION

Marks a position in the script. Not a command — just a named target that branch commands can jump to.

`b`

UNCONDITIONAL BRANCH

Jump to a label immediately. Always fires. If no label given, jumps to end of script.

`t`

BRANCH IF SUBSTITUTION SUCCEEDED

Jump to a label *only* if a successful `s` command has been made since the last line was read or since the last `t`.

GNU sed adds a fourth:

Command	Fires when…	POSIX?
`:label`	Always — it is a target, not a command	Yes
`b label`	Always (unconditional)	Yes
`b` (no label)	Always — jumps to end of script (skips remaining commands)	Yes
`t label`	Only if a `s` command succeeded since last line read or last `t`	Yes
`T label`	Only if NO `s` command has succeeded since last line read or last `t` / `T`	GNU sed only

Labels: `:name`

A label is declared with a colon followed immediately by a name — no space between them:

```
:myloop         # declares a label named "myloop"
:a              # single-letter labels are conventional for brevity
:top            # descriptive names work too
:retry
```

Label names can contain letters, digits, and underscores. Single-letter labels (`:a` , `:b` , `:l`) are the convention in most SED scripts because scripts are often written as compact one-liners. In `-f` script files, longer names improve readability.

> **Labels are not commands.** They produce no output, change nothing in the pattern or hold space, and are completely invisible to the input. They exist only as jump targets for `b` , `t` , and `T` .

The `b` Command — Unconditional Branch

`b label` jumps execution to the named label immediately, regardless of any conditions. It always fires. `b` with no label name jumps to the *end* of the script, which triggers the default print and starts the next cycle.

Skipping the rest of the script

```
# Process lines matching "KEEP" specially; skip all other commands for other lines
sed '/KEEP/! b          # not a KEEP line — skip to end of script (default print)
s/KEEP://              # remove the KEEP: prefix
s/^/  > /             # indent it
' file.txt
```

```
Flow for each line:
/KEEP/! matches? → yes (not a KEEP line) → b fires → jump to end → default print → next cycle → no (IS a KEEP line) → continue: s/KEEP:// → s/^/ > / → print
```

Using `b` in a multi-branch structure

```
# Route lines to different processing branches based on content
sed '/^ERROR/   { s/^/[FAULT]  /; b end }
/^WARNING/ { s/^/[ALERT]  /; b end }
/^INFO/    { s/^/[NOTICE] /; b end }
/^DEBUG/   d
:end' logfile.txt
# Each branch transforms the line and jumps to :end to avoid
# falling through into subsequent branches
```

Jumping forward to skip a block

```
# Skip lines 10–20 from a transformation but still print them
sed '10,20 b skip
s/foo/bar/g
:skip' file.txt
# Lines 10–20: b fires, jumps over s/foo/bar/, still printed by default
# All other lines: s/foo/bar/ runs normally
```

The `t` Command — Branch If Substitution Succeeded

`t` is the conditional branch — it checks an internal flag that SED sets whenever an `s` command makes a successful substitution. If the flag is set, `t label` jumps to the label and **clears the flag**. If the flag is not set (no substitution has succeeded), `t` does nothing.

The flag is also cleared automatically at the start of each new cycle (each new line read) and after each `t` or `T` command fires.

```
t label — decision logic:

Has any s succeeded since last line read or last t/T? → YES: jump to label, clear the flag → NO: do nothing, continue to next command
```

The loop pattern — the most important use of `t`

The most common use of `t` is creating a loop: place a label before a substitution, and after the substitution use `t` to branch back to the label. The loop runs as long as the substitution keeps succeeding:

```
# Remove all leading spaces one at a time (loop until no leading space left)
sed ':loop
s/^ //
t loop' file.txt

# More efficient — but the loop pattern makes the logic visible:
sed 's/^ *//' file.txt  # same result in one step
```

Global replacement with controlled repetition

```
# Replace "aa" with "a" repeatedly until no more double-a pairs remain
sed ':a; s/aa/a/; ta' file.txt
# Input: "aaaa"
# s replaces "aa" → "aaa" (succeeded) → t branches back
# s replaces "aa" → "aa"  (succeeded) → t branches back
# s replaces "aa" → "a"   (succeeded) → t branches back
# s finds no "aa" → fails  → t does NOT branch → end of script
# Output: "a"
```

Joining continuation lines — the classic `t` loop

```
# Collapse lines ending with backslash onto the next line (Makefile/shell style)
sed ':a
/\\$/ {
  N
  s/\\\n//
  ta
}' file.txt
# While the current accumulated line ends with \:
#   N appends the next line
#   s removes the backslash-newline join
#   t loops if the substitution succeeded (i.e. there was a backslash-newline)
```

Removing nested brackets

```
# Remove all content inside parentheses, including nested ones
sed ':a; s/([^()]*)//; ta' file.txt
# [^()]* matches content that contains no brackets (innermost pair)
# Removes the innermost pair each iteration until none remain
# Input:  "result (foo (bar) baz) end"
# Pass 1: s removes "(bar)"       → "result (foo  baz) end"
# Pass 2: s removes "(foo  baz)"  → "result  end"
# Pass 3: s finds nothing to remove → t fails → done
```

Trimming both leading and trailing whitespace in one clean loop

```
sed ':a; s/^[[:space:]]*//; s/[[:space:]]*$//; ta' file.txt
# Loops until neither substitution makes any change
# In practice, both substitutions succeed at most once per line
# — but the loop guarantees correctness even for pathological input
```

The `T` Command — Branch If Substitution Failed

`T` (GNU sed only) is the logical inverse of `t`. It branches to the label if *no* successful substitution has occurred since the last line was read or the last `t` / `T`. Where `t` says "keep looping while things are changing", `T` says "jump away if nothing matched".

```
# Process a line only if a substitution succeeded; skip it otherwise
sed 's/foo/bar/; T skip; s/bar/BAR/
:skip' file.txt
# If s/foo/bar/ succeeded: T does NOT fire, s/bar/BAR/ runs → "BAR"
# If s/foo/bar/ failed:    T fires → jumps to :skip, s/bar/BAR/ never runs
```

```
# Use T to handle "no match" — the else branch of an if-else structure
sed 's/ERROR/FAULT/
T noerror
s/$/  [FLAGGED]/
b done
:noerror
s/$/  [OK]/
:done' logfile.txt
# Lines with ERROR: substitution succeeds → T skips → append [FLAGGED]
# Lines without ERROR: substitution fails → T fires → append [OK]
```

Building If-Else Logic with Branching

SED has no native if-else syntax, but you can construct equivalent logic with addresses, `b`, `t`, and `T`. Here are the standard patterns:

Pattern: if-then (address guard)

```
# "if line matches pattern, do X"
sed '/pattern/ { commands }' file.txt
# The address restricts the block — no branching needed for simple cases
```

Pattern: if-then-else using `b`

```
sed '/pattern/ {
  # THEN branch
  s/foo/bar/
  b done
}
# ELSE branch
s/foo/baz/
:done' file.txt
```

Pattern: if-then-else using `t` / `T`

```
sed 's/pattern/replacement/   # attempt the substitution
T else                         # if it FAILED, jump to else
# THEN branch — substitution succeeded
s/replacement/DONE/
b end
:else
# ELSE branch — substitution failed
s/$/  [no match]/
:end' file.txt
```

Pattern: if NOT using negated address

```
sed '/pattern/! {
  # runs only on lines that do NOT match
  s/foo/bar/
}' file.txt
```

Step-by-Step Traces

Trace: Removing nested parentheses

Script: `sed ':a; s/([^()]*)//; ta'` Input: `sum (x (y+1) z)`

Cycle 1 — PS = "sum (x (y+1) z)" :a label — no action s/([^()]*)// → matches "(y+1)" (innermost, no brackets inside) PS = "sum (x z)" flag SET ta flag is set → branch back to :a, flag cleared :a (back at top) s/([^()]*)// → matches "(x z)" PS = "sum " flag SET ta flag is set → branch back to :a, flag cleared :a (back at top) s/([^()]*)// → no match (no parentheses left) flag NOT set ta flag is not set → does not branch → fall through to end Output: "sum "

Trace: `t` flag reset between cycles

This trace shows the critical point that `t`'s flag is reset with every new line:

Script: sed 's/a/A/; t done; s/b/B/; :done' Input: "ab" / "xy"

Cycle 1 — read "ab" (flag starts clear) s/a/A/ → PS="Ab" flag SET t done → flag set → branch to :done (s/b/B/ is skipped) :done → default print → "Ab" Cycle 2 — read "xy" (flag RESET at start of new cycle) s/a/A/ → no match, PS="xy" flag NOT set t done → flag not set → does NOT branch s/b/B/ → no match, PS="xy" :done → default print → "xy"

Loops in Practice

Converting tabs to spaces (expanding tabs)

```
# Replace each leading tab with 4 spaces, loop until none remain
sed ':a; s/^\t/    /; ta' file.txt
```

Collapsing multiple spaces into one

```
sed ':a; s/  / /; ta' file.txt
# Replaces pairs of spaces with one, loops until no pairs remain
# More efficient equivalent: sed 's/  */ /g' (but the loop form is instructive)
```

Padding a number to fixed width

```
# Left-pad numbers with zeros to 6 digits
sed -E ':a; s/^([0-9]{1,5})$/0\1/; ta' numbers.txt
# Prepends a zero while the number is less than 6 digits long
# 42 → 042 → 0042 → 00042 → 000042 (stops when {1,5} no longer matches)
```

Removing all XML/HTML tags from a file

```
# Handle tags that might span lines (multi-line tags)
sed ':a; s/<[^>]*>//g; /</{ N; ba }' page.html
# First: remove all complete tags on the current line
# If a & lt; remains (incomplete tag), accumulate next line and loop
```

Building a while-loop: process until a condition is met

```
# Keep appending lines until we see a line ending with a semicolon
sed -n ':a
/;$/ { p; d }          # found terminator: print accumulated and delete
N                      # not done yet: append next line
ba' statements.txt
# Accumulates lines until one ends with ; then prints the whole statement
```

Practical Recipes

Comment out lines matching a pattern (idempotent)

```
# Add # to lines matching "debug" only if they are not already commented
sed '/^#/ b            # already a comment — skip
/debug/ s/^/#/' config.sh
```

Replace only the second occurrence of a pattern on each line

```
# Substitute the Nth occurrence cleanly using a loop to skip earlier ones
# Skip the first match (replace it with a marker), replace second, restore first
sed 's/foo/\x00/; s/foo/bar/; s/\x00/foo/' file.txt
# \x00 (null byte) as a temporary marker — won't appear in normal text
```

In-place toggle: switch a config value between two states

```
# Toggle debug=true ↔ debug=false
sed 's/debug=true/debug=TOGGLE/; t done
s/debug=false/debug=true/; t done
s/debug=TOGGLE/debug=false/
:done' app.conf
```

Process only lines in a specific column range

```
# Add a marker if the 5th field of a colon-delimited file is "active"
sed -E 's/^(([^:]*:){4})active:/\1ACTIVE:/' data.txt
# No branching needed here — but a branch pattern lets you act on it further
sed -E 's/^(([^:]*:){4})active:/\1ACTIVE:/
T skip
s/$/ [ENABLED]/
:skip' data.txt
```

Duplicate every line that matches a pattern

```
# Print matching lines twice (original + copy)
sed '/important/ { p; b }
b' file.txt
# p prints first, then b skips to end where default print fires again
# Simpler: sed '/important/p' file.txt (same result)
```

Multi-pass transformation without a second SED invocation

```
# Apply three successive transformations in priority order
# Only the first matching transformation fires (fall-through prevented by b)
sed '/CRITICAL/ { s/CRITICAL/[!!!]/; b }
/ERROR/    { s/ERROR/[!!]/;    b }
/WARNING/  { s/WARNING/[!]/;   b }
/INFO/     { s/INFO/[-]/;      b }' logfile.txt
```

Common Mistakes

Infinite loops

```
# DANGER: this loop never terminates
# s/^/x/ always succeeds (prepending to start of line always works)
sed ':a; s/^/x/; ta' file.txt
# SED will loop forever, growing the line with x's until killed or OOM

# Fix: ensure the loop has a termination condition
# e.g. stop when line starts with 5 x's:
sed ':a; /^xxxxx/! { s/^/x/; ta }' file.txt
```

Misunderstanding `t`'s flag scope

```
# The t flag is shared across ALL s commands in the script for that cycle
# Any successful s anywhere sets the flag — not just the one before t
sed 's/irrelevant/also_irrelevant/  # succeeds on some lines
s/foo/bar/                          # the one we care about
t done                              # fires if EITHER s succeeded!
:done' file.txt
# Fix: use T instead, or restructure so the target s is the only one before t
```

Branch to a non-existent label

```
# Branching to an undefined label is an error in most SED implementations
sed 'b nosuchlabel' file.txt
# sed: can't find label for jump to `nosuchlabel'
# Check spelling — label names are case-sensitive
```

Quick Reference — Chapter 8

LABEL AND BRANCH COMMANDS

`:label`	Declare a label at this point in the script
`b label`	Unconditional jump to *label*
`b`	Unconditional jump to end of script (triggers default print, next cycle)
`t label`	Jump to *label* if any `s` has succeeded since last line read or last t/T
`T label`	Jump to *label* if NO `s` has succeeded (GNU sed only)

KEY PATTERNS

`:a; s/pat/rep/; ta`	Loop: repeat substitution until it stops matching
`/pat/ { cmds; b }`	If-then: run commands and skip rest of script for matching lines
`s/p/r/; T else; cmds; b end; :else; cmds2; :end`	If-else using T (GNU sed)
`/p1/{cmds1;b} /p2/{cmds2;b}`	Multi-branch: route each line to exactly one handler
`:a; /\\$/{N;s/\\\n//;ta}`	Join backslash-continuation lines
`:a; s/([^()]*)//; ta`	Strip nested parentheses iteratively

THE T FLAG — RULES

Set when	Any `s` command makes a successful substitution
Cleared when	A new line is read, or a `t` or `T` command fires
Shared across	All `s` commands in the script — any success sets it

What is coming next: Chapter 9 covers the remaining SED commands — `y` (transliterate), `=` (line number), `r` / `R` (read file), `w` / `W` (write file), `e` (execute), and `z` (zap/clear pattern space). These complete the full SED command set and round out the toolkit before we move on to regular expressions in depth.

CHAPTER 09

The y Command and Other Commands

Chapter 9 — The y Command and Other Commands

Completing the Command Set

The previous chapters covered SED's core commands — `s`, `d`, `p`, `q`, `a`, `i`, `c`, the hold space commands, the multi-line trio, and branching. This chapter covers the remaining commands that round out the full SED toolkit. Some are used daily; others are specialist tools for specific tasks.

`y/src/dst/`

TRANSLITERATE

Map characters one-for-one — every character in *src* is replaced by the corresponding character in *dst*. Works on the entire pattern space, always global.

POSIX standard

`r file`

READ FILE

After printing the current line, read the contents of *file* and insert them into the output stream.

POSIX standard

`R file`

READ ONE LINE FROM FILE

Like `r` but reads only the next unread line from *file*, advancing a file pointer with each call.

GNU sed only

`w file`

WRITE TO FILE

Write the pattern space to *file* immediately. The file is created (or truncated) at script start, then appended to during processing.

POSIX standard

`W file`

WRITE FIRST LINE TO FILE

Like `w` but writes only the first line of the pattern space — useful after multi-line accumulation with `N`.

GNU sed only

`e`

EXECUTE

Execute the pattern space as a shell command and replace it with the command's output. Or (with argument) execute a command without touching pattern space.

GNU sed only

`z`

ZAP (CLEAR)

Clear the pattern space — make it empty without starting a new cycle. Unlike `d`, the rest of the script continues to run.

GNU sed only

`=`

PRINT LINE NUMBER

Print the current input line number to stdout followed by a newline. Introduced in Chapter 4; included here for the complete command reference.

POSIX standard

The `y` Command — Transliterate

The `y` command performs character-by-character translation across the entire pattern space. It works like the standalone `tr` utility but is integrated into SED so it can be combined with addresses and other commands in a single pass.

The syntax mirrors `s` but uses *source* and *destination* character lists instead of a pattern and replacement:

```
y/source-chars/dest-chars/
```

Every character in the pattern space that appears in *source-chars* is replaced by the character at the same position in *dest-chars*. The two lists must be the same length. Like `s`, the delimiter can be changed to any character.

```
Y/ABC/ABC/ — MAPPING
a → A  b → B  c → C  all other chars unchanged
```

Case conversion

```
# Convert lowercase a-z to uppercase A-Z
sed 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' file.txt

# Convert uppercase to lowercase
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/' file.txt

# GNU sed shorthand using character classes in s// is often cleaner for case
sed 's/.*/\U&/' file.txt   # uppercase (GNU sed \U extension)
sed 's/.*/\L&/' file.txt   # lowercase (GNU sed \L extension)
```

ROT13 encoding

```
# Classic ROT13 — rotate letters by 13 positions (encode and decode are identical)
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz/NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefg
```

Transliterating digits and special characters

```
# Replace digits 0-9 with their word equivalents using a proxy
# (y can only map single chars — for multi-char replacements use s//)

# Swap pairs of characters (simple cipher)
sed 'y/aeiouAEIOU/eioauEIOAU/' file.txt

# Replace path separators — Windows to Unix
sed 'y/\\/\//' winpaths.txt
# Backslash → forward slash (each needs escaping)

# Translate specific punctuation
sed 'y/;:,/|||/' data.txt
# Replace semicolons, colons, and commas all with pipe characters
```

y vs s — knowing which to use

	`y/src/dst/`	`s/pattern/replacement/`
Unit of operation	Individual characters	Patterns (strings, regex)
Scope	Always global — entire pattern space	First match only (unless `g` flag)
Replacement length	One char → one char (lists must match in length)	Any length → any length
Regex support	No — source is a literal character list	Yes — full regex in pattern
Special sequences	Only `\n`, `\\`, `\/`	`&`, `\1`–`\9`, `\u`, `\L`, etc.
Best for	Character-set mapping, encoding, case conversion	Everything else

> **y does not support character ranges like `a-z`.** Unlike `tr`, you must list every character explicitly: `y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/`. The `-` character is treated as a literal hyphen in `y`, not a range specifier.

The `r` and `R` Commands — Reading Files

The `r` command reads the entire contents of a named file and inserts them into the output stream after the addressed line. It does not affect the pattern space — the file content is sent directly to output after the current line's normal print completes. If the file does not exist, `r` silently does nothing (no error).

```
r filename       # note: there must be exactly one space between r and the filename
```

Inserting a file after a pattern match

```
# Insert a standard footer after the last line of every file
sed '$r footer.txt' document.txt

# Insert a licence header after line 1 of a script
# (use a after line 1 for before — r always appends after)
sed '1r licence_header.txt' script.sh

# Insert a file after every line matching a pattern
sed '/^## Summary/r summary_boilerplate.txt' report.md
```

Template substitution — the killer use case for `r`

The most powerful use of `r` is template filling: replace a placeholder line with the contents of another file. Combine `r` (to insert the file after the placeholder) with `d` (to delete the placeholder itself):

```
# template.html contains the line: ##CONTENT_PLACEHOLDER##
# Replace it with the contents of content.html
sed '/##CONTENT_PLACEHOLDER##/ {
  r content.html
  d
}' template.html

# Same result, more compact:
sed '/PLACEHOLDER/ { r insert.txt
d }' template.txt
```

> **The order matters:** `r` must come *before* `d` in the block. Since `r` schedules output after the current line's print — and `d` suppresses that print — this combination works correctly: `d` removes the placeholder line from output, while the scheduled file content still appears at that position. If you put `d` first, the `r` would never execute.

The `R` command — reading one line at a time

`R` (GNU sed only) reads a single line from a file each time it fires, advancing an internal pointer. This lets you interleave lines from two files:

```
# Interleave every line of file.txt with lines from extra.txt
sed 'R extra.txt' file.txt
# Output: line1 of file.txt
#         line1 of extra.txt
#         line2 of file.txt
#         line2 of extra.txt  ...

# Substitute a placeholder with one line from a data file each time it appears
sed '/TEMPLATE_LINE/ {
  R data.txt
  d
}' template.txt
```

The `w` and `W` Commands — Writing Files

The `w` command writes the pattern space to a named file. SED creates (or truncates) the output file when the script starts, then appends to it each time `w` fires during processing. The file is ready to use as soon as SED finishes.

```
w filename       # one space between w and filename
```

Splitting a file by content

```
# Route error lines to errors.log and everything else to normal.log
sed -n '/ERROR/  w errors.log
        /ERROR/! w normal.log' app.log

# Split a CSV into separate files by first field value
# (limited — sed can only write to fixed filenames, not dynamic ones)
sed -n '/^alice,/ w alice.csv
/^bob,/   w bob.csv
/^carol,/ w carol.csv' users.csv
```

Writing a subset of lines to a file

```
# Extract all lines containing IP addresses to a separate file
sed -n '/[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}/ w ips.txt' logfile.txt

# Write changed lines to a log while also applying changes in-place
sed -i.bak 's/old_api_url/new_api_url/w changes.log' config.js
# -i.bak edits in place with backup; w flag on s// logs changed lines
```

Using `w` as a side-effect alongside normal output

```
# Print all lines normally, but ALSO write matching lines to a separate file
sed '/ERROR/ w error_summary.txt' app.log
# All lines go to stdout as normal; ERROR lines are additionally written to the file
```

The `W` command — writing only the first line

```
# After N has joined lines, write only the first line of the pair to a file
sed -n '$!N; /^key:/ W keys_only.txt; P; D' data.txt
```

> **File creation behaviour:** SED creates (or truncates to zero) all files named in `w` commands at the very start of execution — even before any input is read. This means if no lines match, the output file still exists, just empty. This is useful to know when checking if a `w` command produced any output: test file size (`-s`) rather than file existence.

The `e` Command — Execute

The `e` command (GNU sed only) has two distinct forms with different behaviours:

Form	Syntax	What it does
As a standalone command	`e command`	Execute *command* in a shell; insert its output into the output stream before the pattern space is printed
As a flag on `s`	`s/pat/rep/e`	After substitution, execute the resulting pattern space as a shell command and replace the pattern space with the command's output

Using `e` as a standalone command

```
# Insert the current date before every line of the file
sed 'e date "+%Y-%m-%d"' file.txt

# Run a command and inject its output before matching lines
sed '/^hostname:/ e hostname' server.conf
# Inserts the actual hostname before each "hostname:" line
```

Using the `e` flag on `s`

```
# For each filename in a list, replace it with its actual file size
sed 's/.*/stat --format="%s bytes: &" &/e' filelist.txt
# s builds a stat command from each filename
# e flag executes that command and replaces the line with its output

# Evaluate each line as a shell arithmetic expression
sed 's/.*/echo $((&))/e' expressions.txt
# Input: "2 + 2"
# s wraps it: "echo $((2 + 2))"
# e executes it → output: "4"
```

> **The `e` command is powerful and dangerous.** It executes arbitrary shell commands using whatever the pattern space contains at that moment. Never use `e` on untrusted input — a crafted input line could execute any command on your system. Reserve it for scripts where you fully control the input data.

The `z` Command — Zap the Pattern Space

The `z` command (GNU sed only) clears the pattern space — sets it to an empty string — without starting a new cycle. This is the key difference from `d`:

	`d` — delete	`z` — zap
Pattern space after	Discarded — cycle ends immediately	Empty string — script continues
Remaining script commands	Skipped entirely	Still execute (on empty pattern space)
Default print	Suppressed	Prints a blank line (empty pattern space)
Use when	You want to remove the line completely	You want to clear the content but still run subsequent commands

```
# Clear the pattern space but append the hold space onto the now-empty line
sed '/HEADER/ { z; G }' file.txt
# z empties the pattern space; G appends the hold space
# Net result: the HEADER line's content is replaced with whatever is in hold space
# (different from c because z + G lets you build the replacement dynamically)

# Use z to reset accumulated content mid-script
sed -n '/START/ { H }
/END/   { H; g; /important/p; z; h }
/START/! { /END/! H }' file.txt
# z resets the pattern space after printing a block
# h then saves the now-empty pattern space back to hold, resetting the accumulator
```

Practical: `z` as a conditional line suppressor

```
# Clear lines that fail a validation check (output blank line as placeholder)
sed '/^[0-9]\+$/ !z' numbers.txt
# Lines that are NOT pure numbers: z clears them (outputs blank line)
# Lines that ARE pure numbers: printed unchanged

# Then pipe to remove the blank lines if you don't want them:
sed '/^[0-9]\+$/ !z' numbers.txt | sed '/^$/d'
# (equivalent to: sed -n '/^[0-9]\+$/p' numbers.txt)
```

The `=` Command — Print Line Number

Already introduced in Chapter 4, `=` prints the current line number to stdout followed by a newline. It is included here for the complete command reference.

```
# Count lines in a file (print only the last line number)
sed -n '$=' file.txt

# Print line numbers alongside matching lines (grep -n equivalent)
sed -n '/ERROR/ { =; p }' logfile.txt

# Add line numbers to every line's output (cat -n equivalent)
sed '=' file.txt | paste - -
# = prints number on its own line; paste merges pairs of lines
```

Combining These Commands — Practical Recipes

Build a simple template engine

```
# template.conf:
#   ServerName ##HOSTNAME##
#   ##SSL_CONFIG##
#   DocumentRoot /var/www/##SITENAME##

sed -e 's/##HOSTNAME##/www.example.com/g' \
    -e 's/##SITENAME##/mysite/g' \
    -e '/##SSL_CONFIG##/ { r ssl_block.conf
                           d }' \
    template.conf
```

Audit log: apply changes and record what changed

```
# Edit in place, write a change log simultaneously
sed -i.bak 's/password=.*/password=REDACTED/w redacted.log' config.ini
# -i.bak: edit in place with backup
# w flag: write lines where substitution succeeded to redacted.log
```

ROT13 encode only comment lines in a script

```
sed '/^#/ y/ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz/NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzab
# y only runs on lines matching /^#/ — code lines are untouched
```

Split output into matched and unmatched files

```
# Write every line to one of two output files, suppress stdout
sed -n '/^[0-9]/ w numeric.txt
       /^[^0-9]/ w alpha.txt' mixed.txt
```

Interleave a file with generated line numbers

```
# Number each line with = and merge onto same line as content
sed '=' file.txt | sed 'N; s/\n/\t/'
# First sed: = prints number on its own line before each content line
# Second sed: N joins pairs; s replaces the joining newline with a tab
```

Evaluate shell expressions embedded in a config file

```
# A config file contains lines like: path=$(pwd)/subdir
# Evaluate only those lines
sed '/\$(/s/.*/echo "&"/e' config.txt
# s wraps matched lines in echo "..."
# e executes: echo evaluates the $() expansion
# WARNING: only use on trusted files
```

Generate a numbered HTML list from plain text

```
# Wrap each line in <li> tags and add ol wrappers
sed -e '1i\<ol>' \
    -e 's/.*\/  <li>&<\/li>/' \
    -e '$a\<\/ol>' \
    items.txt
```

Complete SED Command Reference

Now that all commands have been covered, here is the full command set in one place:

Command	Description	POSIX?
`s/pat/rep/flags`	Substitute	Yes
`d`	Delete pattern space; start next cycle	Yes
`D`	Delete first line of pattern space; restart script	Yes
`p`	Print entire pattern space	Yes
`P`	Print first line of pattern space	Yes
`q [N]`	Print pattern space (unless `-n`), then quit with exit code N	Partial (exit code: GNU)
`Q [N]`	Quit without printing, with optional exit code N	GNU only
`a\text`	Append text after current line	Yes
`i\text`	Insert text before current line	Yes
`c\text`	Replace current line (or range) with text	Yes
`n`	Print pattern space; read next line into it	Yes
`N`	Append next line to pattern space	Yes
`h`	Copy pattern space to hold space	Yes
`H`	Append pattern space to hold space	Yes
`g`	Copy hold space to pattern space	Yes
`G`	Append hold space to pattern space	Yes
`x`	Exchange pattern space and hold space	Yes
`y/src/dst/`	Transliterate characters	Yes
`=`	Print current line number	Yes
`r file`	Read file, insert after current line	Yes
`R file`	Read one line from file, insert after current line	GNU only
`w file`	Write pattern space to file	Yes
`W file`	Write first line of pattern space to file	GNU only
`e [cmd]`	Execute shell command; replace pattern space with output	GNU only
`z`	Clear (zap) the pattern space	GNU only
`:label`	Declare a branch label	Yes
`b [label]`	Branch to label (or end of script)	Yes
`t [label]`	Branch to label if substitution succeeded	Yes

`T [label]`	Branch to label if no substitution succeeded	GNU only

Quick Reference — Chapter 9

TRANSLITERATE — Y

`y/src/dst/`	Replace each char in *src* with the corresponding char in *dst* (always global)
`y/a-z/A-Z/`	Does NOT work — y has no range support; list all chars explicitly

READ AND WRITE FILES

`r file`	Insert entire *file* after current line (POSIX)
`R file`	Insert next unread line of *file* after current line (GNU)
`w file`	Write pattern space to *file* (appending; file created at script start)
`W file`	Write first line of pattern space to *file* (GNU)
`/pat/{r f; d}`	Replace placeholder line with file contents (template pattern)

EXECUTE AND ZAP

`e`	Execute pattern space as shell command; replace with output (GNU)
`e command`	Execute *command*; insert output before current line (GNU)
`s/pat/rep/e`	Substitute then execute result as shell command (GNU)
`z`	Clear pattern space without ending the cycle (GNU)

What is coming next: Chapter 10 goes deep into regular expressions as used specifically in SED — Basic Regular Expressions (BRE) vs Extended Regular Expressions (ERE), character classes, anchors, quantifiers, word boundaries, and the SED-specific quirks and gotchas that catch even experienced users out.

CHAPTER 10

Regular Expressions in SED

Chapter 10 — Regular Expressions in SED

Regex in the SED Context

Regular expressions appear in two places in SED: in **address patterns** (`/regex/command`) and in the **substitute command** (`s/regex/replacement/`). Everything you know about regex applies in both places, but SED has its own flavour — with specific quirks, two distinct modes (BRE and ERE), and some useful extensions that are not found in other tools.

This chapter covers the full regex feature set available in SED: what works in both modes, what requires `-E` , what is GNU-specific, and the common traps that catch even experienced users.

BRE vs ERE — The Two Modes

SED operates in one of two regular expression modes:

- **BRE — Basic Regular Expressions** — the default. Older, more verbose syntax where many metacharacters must be backslash-escaped to activate their special meaning.
- **ERE — Extended Regular Expressions** — activated with the `-E` flag (or `-r` on older GNU sed). Cleaner, more intuitive syntax familiar from `grep -E` , `awk` , and most modern regex flavours.

The table below shows every place where the syntax differs:

Feature	BRE (default)	ERE (with -E)
Grouping	`\( \)`	`( )`
Alternation	`|` (GNU) / not supported (POSIX)	`|`
One or more	`\+` (GNU) / `[a-z][a-z]*` (POSIX)	`+`
Zero or one	`\?` (GNU) / workaround needed (POSIX)	`?`
Repetition exact	`\{n\}`	`{n}`
Repetition range	`\{n,m\}`	`{n,m}`
Backreferences	`\1` … `\9`	`\1` … `\9` (same)
Literal `(`	`(` is literal	`\(` is literal
Literal `+`	`+` is literal	`\+` is literal
Literal `?`	`?` is literal	`\?` is literal
Literal `|`	`|` is literal	`|` is literal

BRE — DEFAULT SED

```
\(foo\|bar\) # group + alternation foo\+ # one or more foo\? # zero or one foo\{3\} # exactly 3 foo\{2,5\} # 2 to 5
```

ERE — SED -E

```
(foo|bar) # group + alternation foo+ # one or more foo? # zero or one foo{3} # exactly 3 foo{2,5} # 2 to 5
```

> **Use `-E` for almost everything.** ERE syntax is cleaner, less error-prone, and more familiar. The only reason to stay in BRE is when writing POSIX-portable scripts that must run on BSD sed without `-E` — even then, test carefully because BSD sed's BRE support has its own quirks.

Anchors

Anchors match positions in the line rather than characters. SED supports four anchors:

Anchor	Matches	Example	Matches
`^`	Start of line (after the implicit newline that precedes the line)	`/^Error/`	Lines beginning with "Error"
`$`	End of line (before the trailing newline stripped by SED)	`/\.$/`	Lines ending with a period
`\b`	Word boundary (GNU sed) — between a word char and a non-word char	`/\bcat\b/`	"cat" but not "concatenate"
`\B`	Non-word boundary (GNU sed) — inside a word	`/\Bcat\B/`	"concatenate" but not standalone "cat"

```
# Match lines that start with a digit
sed -n '/^[0-9]/p' file.txt

# Match blank lines (start immediately followed by end)
sed '/^$/d' file.txt

# Match lines that end with a semicolon
sed -n '/;$/p' source.js

# Replace the word "cat" but not "category" or "concatenate"
sed 's/\bcat\b/dog/g' file.txt

# Remove trailing whitespace (match any whitespace at end of line)
sed 's/[[:space:]]*$//' file.txt
```

> `^` **and** `$` **in multi-line pattern space:** After `N` has joined lines, `^` matches only the very start of the pattern space and `$` matches only the very end. They do *not* match at embedded newlines. To match at embedded line starts/ends use the `m` flag on `s` in GNU sed: `s/^/prefix/mg` — though this is a rarely needed advanced feature.

The Dot — Matching Any Character

`.` matches any single character *except* a newline. This "except newline" rule is the default in SED and most regex flavours. In a multi-line pattern space (after `N`), the embedded `\n` is not matched by `.`.

```
# . matches any single character (not newline)
echo "cat bat hat sat" | sed 's/.at/dog/g'
dog dog dog dog

# Common mistake: using . when you mean a literal dot
echo "192.168.1.1 and 192X168X1X1" | sed 's/192.168/NETWORK/g'
NETWORK.1.1 and NETWORKX1X1   # both matched! . = "any char"

# Fix: escape the dot
echo "192.168.1.1 and 192X168X1X1" | sed 's/192\.168/NETWORK/g'
NETWORK.1.1 and 192X168X1X1   # only the real IP matched

# .* — match everything (greedy)
echo "<b>bold</b> and <b>more</b>" | sed 's/<.*>//'
 and <b>more</b>   # .* is greedy — matches as much as possible
```

Quantifiers

Quantifiers control how many times the preceding element must match:

Quantifier	Meaning	BRE	ERE
Zero or more	Match 0 or more of preceding	`*`	`*`
One or more	Match 1 or more of preceding	`\+` (GNU)	`+`
Zero or one	Match 0 or 1 of preceding (optional)	`\?` (GNU)	`?`
Exactly n	Match exactly *n* times	`\{n\}`	`{n}`
At least n	Match n or more times	`\{n,\}`	`{n,}`
n to m	Match between n and m times (inclusive)	`\{n,m\}`	`{n,m}`

Greedy matching — the default and its consequences

All quantifiers in SED are **greedy** — they match as much as possible while still allowing the overall pattern to succeed. This is the source of many surprising match results:

```
# Greedy: .* matches as much as possible
echo "start middle end" | sed 's/s.*e/X/'
X nd   # matched from "s" all the way to the last "e"

# Lazy matching does not exist in SED (no .*? syntax)
# Workaround: use a negated character class to stop at the first delimiter
echo "<b>bold</b> and <i>italic</i>" | sed 's/<[^>]*>//g'
bold and italic   # [^>]* means "anything that is not >"
```

> **SED has no lazy (non-greedy) quantifiers.** There is no `*?`, `+?`, or `??`. The standard workaround is to use a negated character class `[^delimiter]*` to stop matching at the first occurrence of a boundary character instead of the last.

Practical quantifier examples

```
# Match one or more digits
sed -E 's/[0-9]+/NUM/g' file.txt

# Match an optional minus sign before digits
sed -E 's/-?[0-9]+/NUM/g' file.txt

# Match exactly 4 digits (year)
sed -E 's/\b[0-9]{4}\b/YEAR/g' file.txt

# Match 2 to 4 lowercase letters
sed -E 's/\b[a-z]{2,4}\b/WORD/g' file.txt

# Match an HTTP or HTTPS URL
sed -E 's|https?://[^[:space:]]+|URL|g' file.txt
```

Character Classes

A character class `[...]` matches any single character from the listed set. A negated class `[^...]` matches any character *not* in the set.

Literal character classes

```
# Match any vowel
sed 's/[aeiouAEIOU]/*/g' file.txt

# Match any hex digit
sed -n '/^[0-9A-Fa-f]\+$/p' file.txt

# Match anything that is not a digit
sed 's/[^0-9]//g' file.txt   # strip all non-digits
```

Special characters inside `[...]`

Most regex metacharacters lose their special meaning inside a character class. But a few need care:

Character	Inside `[...]`	How to include literally
`]`	Closes the class	Put it first: `[]abc]`
`-`	Range specifier between two chars	Put it first or last: `[a-z-]` or `[-a-z]`
`^`	Negates the class if first char	Put it anywhere but first: `[a^b]`
`\`	Escape character	`\\`

POSIX character classes — the portable way

POSIX defines named character classes that work correctly across different locales and character sets. These are written inside an extra pair of brackets within the class: `[[:name:]]`.

POSIX class	Equivalent range (ASCII)	Matches
`[[:alpha:]]`	`[a-zA-Z]`	Any letter
`[[:digit:]]`	`[0-9]`	Any digit
`[[:alnum:]]`	`[a-zA-Z0-9]`	Letter or digit
`[[:upper:]]`	`[A-Z]`	Uppercase letter
`[[:lower:]]`	`[a-z]`	Lowercase letter
`[[:space:]]`	`[ \t\n\r\f\v]`	Any whitespace
`[[:blank:]]`	`[ \t]`	Space or tab only
`[[:punct:]]`	(all punctuation)	Any punctuation character
`[[:print:]]`	(all printable)	Any printable character incl. space
`[[:graph:]]`	(printable minus space)	Any printable character excl. space
`[[:cntrl:]]`	(ASCII 0–31, 127)	Control characters
`[[:xdigit:]]`	`[0-9A-Fa-f]`	Hexadecimal digit

```
# Trim leading whitespace using POSIX class (portable across locales)
sed 's/^[[:space:]]*//' file.txt

# Remove all punctuation from a line
sed 's/[[:punct:]]//g' file.txt

# Keep only alphanumeric characters and spaces
sed 's/[^[:alnum:][:space:]]//g' file.txt

# Remove control characters (e.g. stray \r, bell chars)
sed 's/[[:cntrl:]]//g' file.txt
```

Prefer POSIX classes over raw ranges for letters and digits in scripts. `[[:alpha:]]` correctly handles non-ASCII letters when the locale is set to UTF-8, whereas `[a-zA-Z]` may not. For pure ASCII text either works, but POSIX classes are more defensive.

GNU sed Extensions

GNU sed adds several escape sequences not in POSIX BRE or ERE. These are available in both BRE and ERE mode (unless noted):

Sequence	Matches	Equivalent POSIX class
`\w`	Word character	`[[:alnum:]_]`
`\W`	Non-word character	`[^[:alnum:]_]`
`\b`	Word boundary	Position between `\w` and `\W`
`\B`	Non-word boundary	Position inside a word
`\s`	Whitespace character	`[[:space:]]`
`\S`	Non-whitespace character	`[^[:space:]]`
`\d`	Digit (GNU sed 4.9+)	`[[:digit:]]`
`\D`	Non-digit (GNU sed 4.9+)	`[^[:digit:]]`
`\a`	Bell character (ASCII 7)	`[\x07]`
`\t`	Tab character	`[\x09]`
`\n`	Newline (in pattern: embedded newline in pattern space)	—
`\x`*HH*	Character with hex code *HH*	—

```
# Match word boundaries using \b (GNU sed)
sed 's/\bthe\b/THE/g' file.txt

# Strip all whitespace characters (including tabs)
sed 's/\s//g' file.txt

# Match a tab character explicitly
sed 's/\t/  /g' file.txt   # replace tabs with two spaces

# Match word characters (letters, digits, underscore)
sed 's/\w\+/TOKEN/g' file.txt
```

> **GNU extensions are not portable.** Scripts using `\w`, `\s`, `\b`, `\t`, or `\x`*HH* will fail on BSD sed and strict POSIX environments. Use POSIX character classes (`[[:space:]]`, `[[:alnum:]_]`) in scripts that must be portable, and save the GNU extensions for interactive one-liners on Linux systems where you know GNU sed is present.

Grouping and Backreferences

Parentheses group parts of a pattern, and the text matched by each group can be referenced in the replacement string as `\1`, `\2`, etc. This is one of SED's most powerful features.

```
# Capture groups in BRE (backslash required around parentheses)
echo "2024-07-15" | sed 's/\([0-9]\{4\}\)-\([0-9]\{2\}\)-\([0-9]\{2\}\)/\3\/\2\/\1/'
15/07/2024

# Same thing in ERE — much cleaner
echo "2024-07-15" | sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/'
15/07/2024

# Use backreferences to match repeated content (BRE)
echo "the the cat cat" | sed 's/\(\b\w\+\b\) \1/\1/g'
the cat   # removes duplicated consecutive words

# Wrap numbers in brackets
echo "port 8080 and timeout 30" | sed -E 's/([0-9]+)/[\1]/g'
port [8080] and timeout [30]

# Swap two comma-separated fields
echo "Smith,John" | sed -E 's/([^,]+),([^,]+)/\2 \1/'
John Smith
```

Backreferences in patterns (not just replacements)

Backreferences can also appear in the *pattern* itself to match repeated content:

```
# Delete lines where the same word appears twice consecutively
sed '/\(\b[a-z]\+\b\).*\1/d' file.txt

# Delete adjacent duplicate lines (hold space approach — but this also works)
sed '/^\(.*\)$/ { N; /^\(.*\)\n\1$/d }' file.txt
```

Alternation

Alternation matches one of several alternatives. In ERE the pipe character `|` separates alternatives; in BRE you need `\|` (GNU extension — not POSIX).

```
# Match "colour" or "color" — ERE
sed -E 's/colou?r/colour/g' file.txt   # using ? is cleaner here
sed -E 's/colour|color/colour/g' file.txt

# Match any of several log levels
sed -En '/(ERROR|WARN|FATAL)/p' logfile.txt

# Delete lines containing any of three patterns
sed -E '/DEBUG|TRACE|VERBOSE/d' logfile.txt

# BRE equivalent (GNU extension — not portable)
sed '/DEBUG\|TRACE\|VERBOSE/d' logfile.txt
```

```
SED -E 's/(CAT|DOG|BIRD)/ANIMAL/G' APPLIED TO INPUT
I have a cat and a dog and a bird → I have a ANIMAL and a ANIMAL and a ANIMAL The caterpillar → The
ANIMALerpillar # matches within "caterpillar"! Add \b: \b(cat|dog|bird)\b → only whole-word matches
```

Common Regex Traps in SED

Trap 1: Unescaped dot matching too broadly

```
# Intended: match IP 192.168.0.1
sed 's/192.168.0.1/REDACTED/' file.txt
# Problem: matches 192X168Y0Z1 too (. = any char)

# Fix: escape the dots
sed 's/192\.168\.0\.1/REDACTED/' file.txt
```

Trap 2: Greedy .* swallowing too much

```
# Intended: remove each HTML tag individually
echo "<b>bold</b> and <i>italic</i>" | sed 's/<.*>//g'
    # wrong — .* consumed everything from <b> to </i>

# Fix: use [^>]* to stop at the first >
echo "<b>bold</b> and <i>italic</i>" | sed 's/<[^>]*>//g'
bold and italic
```

Trap 3: Forgetting BRE vs ERE parenthesis rules

```
# BRE: ( is literal, \( starts a group
echo "foo(bar)" | sed 's/(bar)/[bar]/'
foo[bar]   # ( and ) are literal in BRE — matched the literal parentheses

echo "foobar" | sed 's/\(bar\)/[\1]/'
foo[bar]   # \( \) are grouping in BRE — captured "bar"

# ERE: ( is grouping, \( is a literal parenthesis
echo "foobar" | sed -E 's/(bar)/[\1]/'
foo[bar]   # ( ) are grouping in ERE

echo "foo(bar)" | sed -E 's/\(bar\)/[bar]/'
foo[bar]   # \( \) are literal in ERE
```

Trap 4: Character range ambiguity in different locales

```
# [a-z] matches differently depending on LC_COLLATE
# In a C/POSIX locale: only a-z ASCII lowercase
# In some UTF-8 locales: may include accented characters between a and z

# Safe: use POSIX class instead
sed 's/[[:lower:]]/*/g' file.txt   # always matches lowercase letters for current locale

# Or force POSIX locale for predictable ASCII behaviour
LC_ALL=C sed 's/[a-z]/*/g' file.txt
```

Trap 5: `*` matching zero occurrences

```
# * means ZERO or more — it can match the empty string
echo "abc" | sed 's/x*/X/g'
XaXbXcX   # x* matches the empty string before and after every character!

# Fix: use + (one or more) instead of * when you need at least one match
echo "abc" | sed -E 's/x+/X/g'
abc   # no match — there are no x's
```

Trap 6: Forgetting that SED regex is not PCRE

```
# These PCRE features do NOT exist in SED:
# (?:...)   — non-capturing group
# (?=...)   — lookahead
# (?<!...)  — lookbehind
# .*?       — lazy quantifier
# \d \w \s  — only in GNU sed, not POSIX
# For complex PCRE needs, use: grep -P, perl, or python
```

Practical Regex Recipes

Validate and extract patterns

```
# Print only valid IPv4 addresses
sed -En '/^([0-9]{1,3}\.){3}[0-9]{1,3}$/p' file.txt

# Print only lines that look like email addresses
sed -En '/^[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}$/p' file.txt

# Extract all hex colour codes from a CSS file
sed -En 's/.*#([0-9A-Fa-f]{3,6}).*/\1/p' styles.css
```

Transformations using groups

```
# Reformat US date MM/DD/YYYY to ISO YYYY-MM-DD
sed -E 's|([0-9]{2})/([0-9]{2})/([0-9]{4})|\3-\1-\2|g' file.txt

# Convert snake_case to camelCase
sed -E 's/_([a-z])/\u\1/g' file.txt
# \u makes next char uppercase (GNU sed replacement extension)

# Add thousands separators to numbers (1234567 → 1,234,567)
sed ':a; s/\([0-9]\)\([0-9]\{3\}\)\b/\1,\2/; ta' file.txt
```

Defensive patterns

```
# Match a whole word safely (not a substring)
sed 's/\bword\b/replacement/g' file.txt

# Match from start of line to first colon (non-greedy via negated class)
sed 's/^[^:]*/KEY/' file.txt   # replace everything before first :

# Match a quoted string (single quotes, no embedded quotes)
sed -E "s/'[^']*'/STRING/g" file.txt
```

Quick Reference — Chapter 10

BRE VS ERE — SYNTAX DIFFERENCES

`-E flag`	Enable ERE — cleaner syntax, recommended for most work
`BRE: \( \) | \+ \? \{n\}`	Grouping, alternation, quantifiers need backslashes
`ERE: ( ) | + ? {n}`	Same features — no backslashes needed

ANCHORS AND SPECIAL SEQUENCES

`^ $`	Start and end of line
`\b \B`	Word boundary / non-boundary (GNU sed)
`\w \W \s \S`	Word / non-word / space / non-space chars (GNU sed)
`\t \n \xHH`	Tab / newline / hex character (GNU sed)

POSIX CHARACTER CLASSES

`[[:alpha:]] [[:digit:]]`	Letters / digits — portable across locales
`[[:alnum:]] [[:space:]]`	Alphanumeric / any whitespace
`[[:upper:]] [[:lower:]]`	Uppercase / lowercase letters
`[[:punct:]] [[:blank:]]`	Punctuation / space+tab only

THE "NO LAZY QUANTIFIER" WORKAROUND

`<.*>`	Greedy — matches from first < to LAST >
`<[^>]*>`	Stops at first > — the lazy equivalent in SED
`[^delimiter]*`	General pattern: match up to (but not including) *delimiter*

What is coming next: Chapter 11 covers SED scripts and real-world recipes — writing multi-command `-f` script files, the essential one-liner library, log file processing, config file manipulation, and CSV/data transformations. Everything learned across the course comes together in practical, ready-to-use patterns.

CHAPTER 11

SED Scripts and Real-World Recipes

Chapter 11 — SED Scripts and Real-World Recipes

From One-Liners to Script Files

So far every example has used the `-e` flag or a single quoted string. For serious work — multi-step transformations, reusable utilities, version-controlled text processing tools — you write a **SED script file** and invoke it with `sed -f scriptfile.sed input.txt`.

A script file is plain text. Each line is a SED command, exactly as you would write after `-e`, but without the quoting and with full freedom to use whitespace, blank lines, and comments.

```
ANATOMY OF A SED SCRIPT FILE (CLEANUP.SED)
#!/usr/bin/sed -f                          optional shebang — makes it self-executable
# ── normalise_config.sed ──────────────────  comment: whole line starting with #
# Remove blank lines and comments          describe intent, not mechanics
/^[[:space:]]*$/d                          delete blank lines
/^[[:space:]]*#/d                          delete comment lines
                                           blank lines are fine — improve readability
# Trim trailing whitespace
s/[[:space:]]*$//                          strip trailing spaces / tabs
# Normalise multiple spaces to one
s/[[:space:]]\+/ /g                        collapse runs of whitespace
```

Invoking a script file

```
# Basic usage
sed -f normalise_config.sed input.conf

# In-place edit
sed -i -f normalise_config.sed *.conf

# Combine a script file with an extra -e command
sed -f normalise_config.sed -e 's/localhost/10.0.0.1/g' input.conf

# ERE mode with a script file
sed -E -f myscript.sed input.txt

# Self-executable script (shebang must be first line)
chmod +x normalise_config.sed
./normalise_config.sed input.conf
```

> **Comments in script files vs one-liners:** The `#` comment character works in `-f` script files. In a `-e` expression, `#` is *not* a comment — it is a literal character or part of a regex delimiter. Only use comments in script files.

Structuring Complex Scripts

Using braces for clarity

```
# Group related commands under an address — visually clear in script files
/^[[:space:]]*\[/ {
    # Lines that look like INI section headers
    s/[[:space:]]//g        # strip all spaces
    s/\[/[/             # rewrite opening bracket (noop here but explicit)
    y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/  # uppercase
}

# Process lines NOT in the header range
/^[[:space:]]*\[/! {
    s/^[[:space:]]*//       # trim leading whitespace
    s/[[:space:]]*$//       # trim trailing whitespace
}
```

Labels and branching for loops in scripts

```
# Script: collapse repeated blank lines into a single blank line
# Algorithm: join current line to next with N, check for blank+blank,
# loop until no longer two consecutive blanks.

/^$/ {
    N
    /^\n$/b blankloop   # if still blank after join, keep looping
    P
    D
}
b                       # non-blank line: fall through to default print

:blankloop
N
/^\n$/b blankloop
P
D
```

Log File Processing

Real-World Recipe Category 1

Filter and extract from logs

```
# Show only ERROR and FATAL lines (suppress everything else)
sed -n '/\(ERROR\|FATAL\)/p' app.log

# Same in ERE
sed -En '/(ERROR|FATAL)/p' app.log

# Delete DEBUG and TRACE lines in-place (quiet the noise)
sed -i '/\(DEBUG\|TRACE\)/d' app.log

# Anonymise IP addresses (replace last two octets)
sed -E 's/([0-9]{1,3}\.[0-9]{1,3})\.[0-9]{1,3}\.[0-9]{1,3}/\1.x.x/g' access.log

# Extract just the timestamp column (first field before space)
sed 's/[[:space:]].*//' app.log

# Add a prefix to every log line for pipeline tagging
sed 's/^/[APP] /' app.log
```

Reformat Apache / Nginx combined log format

```
# Combined log format:
# IP - - [timestamp] "METHOD /path HTTP/1.1" status bytes
# Extract: IP, status, path
sed -E 's/^([^ ]+) [^ ]+ [^ ]+ \[[^]]+\] "[^ ]+ ([^ ]+)[^"]*" ([0-9]+).*/\1 \3 \2/' access.log

# Count each unique status code (pipe to sort | uniq -c for frequency)
sed -E 's/^.* "[^"]*" ([0-9]{3}) .*/\1/' access.log | sort | uniq -c

# Keep only 4xx and 5xx errors
sed -En '/\" [45][0-9]{2} /p' access.log
```

Multi-line log entry processing

```
# Java stack traces span multiple lines — collect them
# Strategy: when a line starts with "at " or "Caused by:", append it
# to the previous line so the full trace stays together

# Simple version: join lines beginning with whitespace (indented continuations)
sed '/^[[:space:]]/{H;d}; /^[[:space:]]/!{x;s/\n/ /g;p}' -n app.log
```

Config File Manipulation

Real-World Recipe Category 2

INI / key=value files

```
# Set a specific key to a new value (key=value format)
sed 's/^\(max_connections\s*=\s*\).*/\1100/' db.conf

# Same with ERE — cleaner
sed -E 's/^(max_connections\s*=\s*).*/\1100/' db.conf

# Comment out a specific key (prepend #)
sed 's/^\(listen_address\s*=\)/#\1/' db.conf

# Uncomment a specific key (remove leading #)
sed 's/^#\(listen_address\s*=\)/\1/' db.conf

# Delete a key entirely
sed '/^old_deprecated_key\s*=/d' db.conf

# Append a new key after an existing one
sed '/^max_connections/a new_param = 50' db.conf

# Insert a key before an existing one
sed '/^\[database\]/i # Database section begins here' db.conf
```

Work safely within a specific section only

```
# Only modify keys inside the [database] section
# Strategy: range from section header to the next section header (or EOF)
sed -E '/^\[database\]/,/^\[/ s/^(host\s*=\s*).*/\1newhost/' app.conf

# Caution: the closing /^\[/ also matches the NEXT section header line itself
# To avoid modifying that line, negate it:
sed -E '/^\[database\]/,/^\[/ { /^\[database\]/! { /^\[/! s/^(host\s*=\s*).*/\1newhost/ } }' app.conf
```

XML / HTML attribute updates

```
# Update a version attribute in a single-line XML tag
sed -E 's/(version=")[^"]*/\11.2.3/' pom.xml

# Replace a URL inside quotes (uses | as delimiter to avoid escaping /)
sed -E 's|href="[^"]*"|href="https://new.example.com"|g' index.html

# Remove a specific HTML attribute entirely
sed -E 's/ style="[^"]*"//g' index.html
```

CSV and Tabular Data

Real-World Recipe Category 3

SED is line-oriented and not CSV-aware. It does not understand quoted fields that contain commas or newlines. For production CSV with embedded commas, use `awk`, Python's `csv` module, or `miller`. The recipes below work reliably only on simple delimiter-separated data with no embedded delimiters.

```
# Extract the third comma-separated field
sed -E 's/^([^,]*,){2}([^,]*).*/\2/' data.csv

# Delete the first field (up to and including the first comma)
sed 's/[^,]*,//' data.csv

# Replace comma delimiter with pipe
sed 's/,/|/g' data.csv

# Add a header line if the file lacks one
sed '1i name,age,city' data.csv

# Skip the header line and process only data rows
sed '1d; s/,/ /g' data.csv

# Surround every field with double quotes (simple CSV → quoted CSV)
sed -E 's/([^,]+)/"\1"/g' data.csv

# Trim spaces around comma separators
sed 's/[[:space:]]*,[[:space:]]*/,/g' data.csv
```

Source Code Transformations

Real-World Recipe Category 4

Rename a function or variable across a codebase

```
# Safe rename: only match whole identifiers (not substrings)
sed -i 's/\bgetUser\b/fetchUser/g' src/*.js

# Rename across all .py files recursively (using find + xargs)
find . -name '*.py' | xargs sed -i 's/\bold_func\b/new_func/g'
```

Add / remove language constructs

```
# Add a semicolon to every non-blank, non-comment line (JS minification prep)
sed '/^[[:space:]]*$/d; /^[[:space:]]*\/\//d; s/$/;/' code.js

# Remove C-style single-line comments (// to end of line)
sed 's|//.*$||' code.c

# Add "export " prefix to every function declaration
sed '/^function /s/^/export /' module.js

# Indent every line by 4 spaces (e.g. when inserting into another file)
sed 's/^/    /' snippet.py

# Remove all indentation (dedent)
sed 's/^[[:space:]]*//' snippet.py

# Convert tabs to 4 spaces
sed 's/\t/    /g' code.py

# Remove trailing whitespace (common pre-commit hook fix)
sed -i 's/[[:space:]]*$//' *.py
```

Version number bumping

```
# Bump patch version X.Y.Z → X.Y.Z+1 is tricky without arithmetic
# But you can replace a known version string precisely:
sed 's/version = "1.2.3"/version = "1.2.4"/' pyproject.toml

# Replace any version string matching the pattern (capture major.minor, replace patch)
# Note: SED cannot do arithmetic — only string replacement
sed -E 's/(version = ")([0-9]+\.[0-9]+\.)[0-9]+/\11.2.4/' pyproject.toml
```

Text Formatting and Document Processing

Real-World Recipe Category 5

Line spacing

```
# Double-space a file (add a blank line after every line)
sed 'G' file.txt

# Remove all blank lines
sed '/^[[:space:]]*$/d' file.txt

# Remove consecutive blank lines (leave at most one blank line)
sed '/./,/^$/!d' file.txt   # classical one-liner

# Number every line
sed '=' file.txt | sed 'N; s/\n/\t/'

# Number only non-blank lines
sed '/./=' file.txt | sed '/./N; s/\n/\t/'
```

Reverse a file (tac equivalent)

```
# Pure SED tac: accumulate all lines in hold space, then print reversed
sed -n '1!G; h; $p' file.txt
# 1!G  — on every line except the first: prepend hold space to pattern space
# h    — copy pattern space to hold space
# $p   — on the last line: print (now contains all lines reversed)
```

Prepend and append boilerplate

```
# Insert a copyright header at the very top of a file
sed -i '1i # Copyright 2024 Example Corp. All rights reserved.' *.py

# Append a footer line at the very end
sed -i '$a # End of file' file.txt

# Replace lines 1-3 (existing header) with a new header
sed -i '1,3c\# New header line 1\n# New header line 2' file.txt

# Wrap entire file content in XML tags
sed '1i <document>; $a </document>' content.txt
```

Markdown and plain-text utilities

```
# Convert Markdown headings to HTML
sed -E 's/^### (.+)/<h3>\1<\/h3>/;
         s/^## (.+)/<h2>\1<\/h2>/;
         s/^# (.+)/<h1>\1<\/h1>/' doc.md

# Convert *word* to <em>word</em>
sed -E 's/\*([^*]+)\*/<em>\1<\/em>/g' doc.md

# Remove all Markdown emphasis markers (strip * and _)
sed 's/[*_]//g' doc.md

# Centre-align text within 72 characters (approximate)
sed 's/^/                                    /; s/^\(.\{36\}\)\(.*\)/\1\2/' title.txt
```

Data Masking and Security

Real-World Recipe Category 6

```
# Mask credit card numbers (keep last 4 digits)
sed -E 's/\b([0-9]{4}[- ]?){3}([0-9]{4})\b/****-****-****-\2/g' report.txt

# Redact email addresses
sed -E 's/[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}/[REDACTED]/g' data.txt

# Mask all but the first and last character of a password field
sed -E 's/(password=")([^"])(.*)(.")(\s*")/\1\2***\4\5/g' config.xml

# Remove any line containing an API key pattern
sed -i '/api[_-]key\s*[:=]/Id' config.txt   # I flag = case-insensitive (GNU)

# Scrub phone numbers (various formats)
sed -E 's/(\+?1[-. ]?)?(\(?\d{3}\)?[-. ]?)(\d{3}[-. ]?\d{4})/[PHONE]/g' data.txt
```

The Essential SED One-Liner Library

These are the most referenced SED one-liners — memorise or bookmark them:

DELETE BLANK LINES

```
sed '/^$/d'
```

REMOVE TRAILING WHITESPACE

```
sed 's/[[:space:]]*$//'
```

REMOVE LEADING WHITESPACE

TRIM BOTH ENDS

```
sed 's/^[[:space:]]*//'
```

```
sed 's/^[[:space:]]*//; s/[[:space:]]*$//'
```

PRINT LINES 5 TO 10

```
sed -n '5,10p'
```

PRINT LAST LINE

```
sed -n '$p'
```

DOUBLE-SPACE

```
sed 'G'
```

NUMBER LINES

```
sed '=' | sed 'N;s/\n/\t/'
```

REVERSE FILE (TAC)

```
sed -n '1!G;h;$p'
```

PRINT BETWEEN PATTERNS

```
sed -n '/START/,/END/p'
```

DELETE BETWEEN PATTERNS

```
sed '/START/,/END/d'
```

REMOVE DUPLICATE ADJACENT LINES

```
sed '$!N;/^\(.*\)\n\1$/!P;D'
```

IN-PLACE BACKUP EDIT

```
sed -i.bak 's/foo/bar/g' file
```

PREPEND TEXT TO EVERY LINE

```
sed 's/^/PREFIX: /'
```

CONVERT WINDOWS LINE ENDINGS

```
sed 's/\r$//'
```

REPLACE NTH OCCURRENCE ON A LINE

```
sed 's/foo/bar/2'  # 2nd only
```

Building Reusable SED Libraries

For teams or personal toolkits, it pays to organise frequently used SED scripts as self-documenting files:

```
#!/usr/bin/sed -Ef
# ──────────────────────────────────────────────────
# sanitise_csv.sed
# Purpose : Normalise a messy CSV before import
# Usage   : sed -Ef sanitise_csv.sed input.csv > output.csv
# Requires: GNU sed (uses \s, -E, -f together)
# ──────────────────────────────────────────────────

# 1. Remove UTF-8 BOM if present (first line, first three bytes = EF BB BF)
1s/^\xef\xbb\xbf//

# 2. Normalise Windows CRLF to LF
s/\r$//

# 3. Trim whitespace around each comma
s/[[:space:]]*,[[:space:]]*/,/g

# 4. Remove trailing comma if present
s/,$//

# 5. Remove completely empty lines
/^$/d
```

Good script file habits: Always include a usage comment, document the GNU/POSIX requirements, group related commands with a blank line and comment, and name script files with a `.sed` extension so editors apply syntax highlighting.

Quick Reference — Chapter 11

SCRIPT FILE INVOCATION

`sed -f script.sed file`	Run a SED script file against a file
`sed -i -f script.sed file`	In-place edit using a script file
`sed -E -f script.sed file`	ERE mode with a script file
`sed -f s1.sed -f s2.sed file`	Chain multiple script files
`sed -f script.sed -e 's/x/y/'`	Script file plus inline expression

SCRIPT FILE BEST PRACTICES

`# comment text`	Whole-line comments — only work in -f files, not -e
`#!/usr/bin/sed -f`	Shebang for self-executable scripts (first line only)
`Blank lines between groups`	Ignored by SED — use them freely for readability
`.sed extension`	Convention — enables editor syntax highlighting

What is coming next: Chapter 12 — the final chapter — covers SED in shell scripts and pipelines: building robust wrappers, using SED inside functions, combining SED with grep/awk/cut in production pipelines, performance on large files, error handling, and when to stop using SED and reach for a different tool.

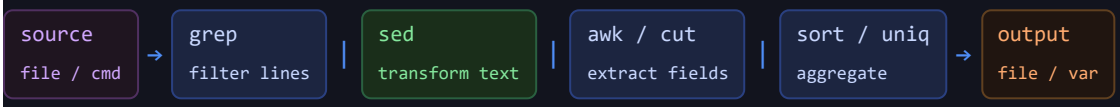
CHAPTER 12

SED in Shell Scripts and Pipelines

Chapter 12 — SED in Shell Scripts and Pipelines

SED as a Pipeline Stage

SED's greatest strength is that it reads from stdin and writes to stdout — it fits naturally anywhere in a Unix pipeline. Every tool in the pipeline handles one concern, and SED handles the text transformation step.




```
# Typical pipeline: filter → transform → aggregate
grep "ERROR" app.log \
  | sed -E 's/.*\[([0-9:T-]+)\].*/\1/' \
  | sort | uniq -c | sort -rn

# SED reading from a command's output via process substitution
sed 's/foo/bar/g' <(curl -s "https://example.com/data.txt")

# SED writing to a variable via command substitution
CLEANED=$(sed 's/[[:space:]]*$//' <<< "  hello world  ")
echo "'$CLEANED'"
'  hello world'
```

Using SED Inside Shell Functions

Wrapping SED in shell functions creates readable, reusable utilities. The function handles argument validation and quoting; SED handles the transformation.

Basic wrapper pattern

```
# Function: trim whitespace from both ends of a string
trim() {
    sed 's/^[[:space:]]*//; s/[[:space:]]*$//' <<< "$1"
}

# Usage
trim "   hello world   "
hello world

# Capture the result
RESULT=$(trim "  data  ")
```

Functions that process files or stdin

```
# Function: strip all comments and blank lines from a config file
strip_config() {
    local file="${1:--}"   # default to stdin if no arg
    sed '/^[[:space:]]*#/d; /^[[:space:]]*$/d' "$file"
}

# Use on a file
strip_config /etc/ssh/sshd_config

# Use on stdin via pipeline
cat app.conf | strip_config

# ──────────────────────────────────────────────────
# Function: replace a key=value in a file (safe in-place)
set_config_key() {
    local key="$1" value="$2" file="$3"
    if [[ -z "$key" || -z "$file" ]]; then
        echo "Usage: set_config_key KEY VALUE FILE" >&2
        return 1
    fi
    # Escape the value so it's safe inside the sed replacement
    local escaped_value
    escaped_value=$(sed 's/[\/&]/\\&/g' <<< "$value")
    sed -i "s|^\($key\s*=\s*\).*|\1$escaped_value|" "$file"
}

# Usage
set_config_key "max_connections" "200" db.conf
set_config_key "host" "db.example.com" app.conf
```

Passing shell variables into SED expressions

```
# Use double quotes to expand shell variables in the sed expression
OLD="localhost"
NEW="10.0.0.5"
sed "s/$OLD/$NEW/g" config.txt

# Problem: if variables contain / you need to escape or change delimiter
OLD_URL="http://old.example.com/api"
NEW_URL="https://new.example.com/v2"
sed "s|$OLD_URL|$NEW_URL|g" config.txt   # use | as delimiter

# Safer: escape the variable content before inserting into the pattern
escape_sed_pattern() {
    # Escapes special BRE characters in a string for use in a sed pattern
    sed 's/[]\/$*.^[]/\\&/g' <<< "$1"
}
escape_sed_replacement() {
    # Escapes special replacement characters (& and \)
    sed 's/[\/&]/\\&/g' <<< "$1"
}

# Safe substitution using the escape helpers
PATTERN=$(escape_sed_pattern "$OLD_URL")
REPLACE=$(escape_sed_replacement "$NEW_URL")
sed "s|$PATTERN|$REPLACE|g" config.txt
```

> **Never use single quotes when you need variable expansion, and never use double quotes when the pattern contains characters the shell will misinterpret (like `!` in history-enabled shells).** For complex patterns containing both shell metacharacters and SED special characters, build the expression in a variable first, then pass it.

Safe In-Place Editing Patterns

In-place editing with `-i` is powerful but destructive if something goes wrong. In scripts you should always protect against failures.

```bash
# Always take a backup with -i.bak before modifying in scripts
sed -i.bak 's/foo/bar/g' important.conf

# Verify success, restore on failure
if ! sed -i.bak 's/foo/bar/g' important.conf; then
    echo "sed failed — restoring backup" >&2
    mv important.conf.bak important.conf
    exit 1
fi

# Atomic edit pattern: write to temp file, then move (POSIX mv is atomic)
atomic_sed_edit() {
    local expr="$1" file="$2"
    local tmp
    tmp=$(mktemp)
    if sed "$expr" "$file" > "$tmp"; then
        mv "$tmp" "$file"
    else
        rm -f "$tmp"
        echo "ERROR: sed edit failed on $file" >&2
        return 1
    fi
}

# Usage
atomic_sed_edit 's/version=1\.0/version=2.0/' app.conf
```

Why atomic? With `sed -i`, if the process is killed mid-write, the file is left partially written and the original is already gone. The temp-file-then-move pattern keeps the original intact until the new version is fully written. On the same filesystem, `mv` is a single inode rename — instantaneous and atomic.

Error Handling and Exit Codes

```
# SED exit codes
# 0 — success (even if no substitutions were made)
# 1 — error (bad syntax, file not found)
# 2 — usage error (GNU sed)

# Check if sed succeeded
if sed -n '/pattern/p' file.txt > output.txt; then
    echo "Processing complete"
else
    echo "sed encountered an error" >&2
    exit 1
fi

# Note: sed returns 0 even if no lines matched — it is not grep
# To detect whether a substitution actually happened, use the w flag + wc -l
COUNT=$(sed -n 's/foo/bar/w /dev/stdout' file.txt | wc -l)
echo "$COUNT lines were changed"

# Validate a sed expression before using it on real data
if echo "" | sed -n "$EXPR" 2>/dev/null; then
    echo "Expression is valid"
else
    echo "Invalid sed expression: $EXPR" >&2
    exit 1
fi
```

Performance on Large Files

SED is fast — it processes line by line with minimal memory overhead. But there are patterns that hurt performance, and strategies to avoid them.

What slows SED down

Anti-pattern	Problem	Fix
`sed 'complex/pattern/' huge.log`	Applying expensive patterns to every line	Pre-filter with `grep -F` to reduce line count first
`sed -i 's/x/y/' file` in a loop	Opens and rewrites the file once per iteration	Combine all substitutions into one `sed` call or a script file
Chained `sed | sed | sed`	Multiple processes, multiple passes through the data	Merge into one `sed` call with multiple `-e` or a script file
`sed 'N; P; D'` on huge files	Multi-line commands load two lines at once — still fine, but beware `N` on the last line	Use `$!N` to protect the last line
`sed '1,/pattern/{...}'` range that never closes	Every line after line 1 is tested against the regex forever	Ensure the closing pattern will actually be found, or use line numbers

Practical performance patterns

```
# BAD: three separate sed processes
sed 's/foo/bar/g' data.txt | sed 's/baz/qux/g' | sed '/^#/d'

# GOOD: one process, three expressions
sed 's/foo/bar/g; s/baz/qux/g; /^#/d' data.txt

# BEST for large files: pre-filter then transform
grep -F "foo" huge.log | sed 's/foo/bar/g'

# Use q/Q to stop early when you only need the first match
sed -n '/pattern/{p;q}' huge.log   # stop at first match — don't read the rest

# Process only a known range to avoid scanning the whole file
sed -n '1000,2000p' huge.log   # print lines 1000-2000 then stop

# For enormous files: use LC_ALL=C to avoid multibyte locale overhead
LC_ALL=C sed 's/foo/bar/g' 10gb.log
```

SED with find, xargs and Loops

```
# Edit all .conf files in a directory tree in-place
find /etc -name '*.conf' -exec sed -i.bak 's/old/new/g' {} \;

# Faster: use xargs to batch files (fewer sed invocations)
find /etc -name '*.conf' | xargs sed -i.bak 's/old/new/g'

# Handle filenames with spaces safely
find . -name '*.txt' -print0 | xargs -0 sed -i 's/foo/bar/g'

# Shell loop with sed — useful when you need per-file logic
for file in src/*.js; do
    if grep -q "TODO" "$file"; then
        echo "Processing: $file"
        sed -i 's/TODO/FIXME/g' "$file"
    fi
done

# Edit files modified in the last 24 hours
find . -name '*.py' -mtime -1 | xargs -0r sed -i 's/\t/    /g'
```

Combining SED with Other Tools

Knowing when to hand off to a different tool is as important as knowing how to use SED. Here is a practical guide to the most common partnerships:

Task	Best tool	Why not just sed?
Simple pattern filter	grep	Faster, simpler, returns correct exit code on match/no-match
Field extraction by column	cut / awk	SED regex for fields gets unwieldy quickly
Arithmetic on values	awk / bc	SED has no arithmetic capability
Sorting, deduplication	sort / uniq	SED can't sort — it is strictly ordered line processing
Structured data (JSON/YAML)	jq / yq / python	SED treats everything as text — will break on nested structures
CSV with quoted fields	awk / python csv / miller	SED cannot handle embedded commas or newlines in quoted fields
Complex multi-field transforms	awk	awk has variables, arrays, arithmetic, printf — SED has none
Line-by-line text substitution	sed	This is exactly what SED is built for
Multi-line pattern matching	sed (with N/P/D) or perl	Doable in SED but verbose; perl -0777 is simpler for whole-file
In-place file editing	sed -i	Perl -i also works; most other tools require temp files

Classic SED + awk pipeline

```
# SED strips noise, awk does the field work and arithmetic
grep "RESPONSE_TIME" app.log \
  | sed -E 's/.*RESPONSE_TIME=([0-9.]+)ms.*/\1/' \
  | awk '{ sum+=$1; n++ } END { printf "avg: %.2fms\n", sum/n }'

# SED normalises the format, cut extracts the column
sed 's/  */ /g' data.txt | cut -d ' ' -f3

# sed + sort + uniq for a frequency count
sed -E 's/.*status=([0-9]{3}).*/\1/' app.log | sort | uniq -c | sort -rn
```

Portability Checklist for Scripts

If your script must run on both Linux (GNU sed) and macOS (BSD sed), test against this checklist:

Feature	GNU sed	BSD sed (macOS)	Safe alternative
`-i` in-place	-i ''	Requires -i '' (empty string arg)	`sed -i ''` (works on both if empty string given)
`-E` ERE flag	-E or -r	-E	Use `-E` (not `-r`)
`\+` `\?` in BRE	Supported	Not supported	Use `-E` with `+` `?`
`|` alternation BRE	Supported	Not supported	Use `-E` with `|`
`\w` `\s` `\b`	Supported	Not supported	Use `[[:alnum:]_]` etc.
`\t` in regex	Supported	Not supported	Use `$'\t'` or POSIX `[[:blank:]]`
`a\` append	Both forms	POSIX backslash form only	Use `a\ text` (backslash-newline)
`T` branch-if-not-substituted	Supported	Not supported	No portable equivalent — use `t` with negation
`z` zap pattern space	Supported	Not supported	Use `s/.*//` or `d`
`R` `W` commands	Supported	Not supported	No portable equivalent

```
# Portable in-place edit that works on both GNU and BSD sed
if sed --version 2>/dev/null | grep -q 'GNU'; then
    # GNU sed: -i accepts no argument (or empty string)
    sed -i 's/foo/bar/g' file.txt
else
    # BSD sed: -i requires an explicit extension argument (even if empty)
    sed -i '' 's/foo/bar/g' file.txt
fi
```

When to Stop Using SED

SED is the right tool for fast, line-oriented text transformations. Reach for something else when:

- **The data has structure** (JSON, YAML, XML, real CSV) — use a parser that understands the format. SED treats everything as flat text.
- **You need arithmetic** — SED cannot add, subtract, or compare numbers. Use `awk` or `bc`.
- **The regex would need lookahead/lookbehind** — SED has no PCRE. Use `grep -P`, `perl`, or `python`.
- **The script exceeds ~20 lines** — a SED script of 30+ lines is a maintenance burden. Rewrite in Python or awk where logic can be clearly expressed.
- **You need error handling, conditionals, or data structures** — SED has none of these in any meaningful way. Use a real scripting language.
- **The transformation depends on values from other lines** (e.g. database-style joins, lookups) — SED's hold space is a single buffer; use awk or a scripting language.

The SED sweet spot: one-liners and short scripts performing text substitution, line filtering, or simple reformatting on line-oriented input. Anything beyond that — reach for awk first, then Python. Use SED where it is fast and clear; stop before it becomes a puzzle.

Quick Reference — Chapter 12

SED IN PIPELINES

`cmd | sed 'expr'`	Transform output of a command
`sed 'expr' <(cmd)`	Process substitution — treat command output as a file
`VAR=$(sed 'expr' <<< "$STR")`	Capture sed output into a variable
`LC_ALL=C sed 'expr' file`	Force C locale for maximum speed on ASCII files

SAFE SCRIPTING HABITS

`sed -i.bak 'expr' file`	Always backup before in-place edit in scripts
`tmp=$(mktemp); sed … > "$tmp" && mv "$tmp" file`	Atomic edit — original safe until new version is complete
`sed "s/$VAR/…/"`	Double-quote to expand variables; escape / with | delimiter
`find … | xargs -0 sed -i`	Null-delimited xargs handles filenames with spaces
`grep -F 'literal' | sed 'expr'`	Pre-filter with grep on large files for a big speed gain

Course Complete — Full Chapter Index

✓

SED Course Complete

All 12 chapters complete. You now have a thorough understanding of SED — from its processing model and core commands through to real-world scripting, pipeline integration, and performance-aware usage.