

Text Search Commands

LinuXpert Systems

No.5, Dr. Ambedkar St, Rathinapuram

Nanganallur, Chennai – 600 061.

Phone: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. grep 1

2. egrep 9

3. fgrep 16

4. awk 21

5. sed 31

6. diff 41

7. cmp 48

1. `grep` (Global Regular Expression Print)

`grep` is a Linux/Unix command used to **search for text patterns** in files or command output. It prints the lines that match the specified pattern.

Syntax

```
grep [OPTIONS] PATTERN [FILE...]
```

- **PATTERN** – The text or regular expression to search for.
- **FILE** – One or more files to search.

Basic Examples

1. Search for a word in a file

```
grep "error" logfile.txt
```

Output

```
Error: Disk full
error: permission denied
```

Searches for the word **error** in `logfile.txt`.

---

2. Search multiple files

```
grep "main" file1.c file2.c
```

Searches for `main` in both files.

---

3. Ignore case

```
grep -i "linux" file.txt
```

Matches:

```
Linux

LINUX

linux

LiNuX
```

---

4. Show line numbers

```
grep -n "error" logfile.txt
```

Example:

```
12:error: file not found
45:error: invalid option
```

---

5. Count matching lines

```
grep -c "failed" auth.log
```

Output:

```
18
```

---

6. Invert the match

```
grep -v "^#" config.conf
```

Displays all lines **except** those beginning with #.

---

7. Recursive search in directories

```
grep -r "TODO" .
```

Searches every file in the current directory and subdirectories.

---

8. Match whole words only

```
grep -w "cat" animals.txt
```

Matches:

```
cat
```

Does **not** match:

```
catalog
scatter
```

---

9. Exact line match

```
grep -x "hello" file.txt
```

Matches only lines containing exactly:

```
hello
```

---

10. Show only matching text

```
grep -o "[0-9]\+" file.txt
```

If the file contains:

```
ID: 12345
```

Output:

```
12345
```

Common Options

Option	Description
`-i`	Ignore case
`-v`	Show non-matching lines
`-n`	Display line numbers
`-c`	Count matching lines
`-r` or `-R`	Search directories recursively
`-l`	Show filenames with matches
`-L`	Show filenames without matches
`-w`	Match whole words only
`-x`	Match entire line
`-o`	Print only the matching text
`-A N`	Show N lines after the match
`-B N`	Show N lines before the match
`-C N`	Show N lines before and after the match
`-E`	Use extended regular expressions
`-F`	Search for fixed strings (faster, no regex)
`-e`	Specify multiple search patterns
`-f`	Read patterns from a file
`--color=auto`	Highlight matches

Using Regular Expressions

Match lines starting with "root"

```
grep "^root" /etc/passwd
```

Match lines ending with "bash"

```
grep "bash$" /etc/passwd
```

Match any digit

```
grep "[0-9]" file.txt
```

Match lowercase letters

```
grep "[a-z]" file.txt
```

Match multiple patterns

```
grep -E "error|warning" logfile.txt
```

Matches either:

```
error
warning
```

Using `grep` with Pipes

Search running processes:

```
ps -ef | grep firefox
```

Search network connections:

```
ss -tulpn | grep ssh
```

Search command history:

```
history | grep apt
```

Search installed packages:

```
dpkg -l | grep nginx
```

Search Recursively for a File Type

Search only `.c` files:

```
grep "printf" *.c
```

Search recursively:

```
grep -rn "printf" src/
```

Exclude Files

Exclude `.log` files:

```
grep -r --exclude="*.log" "error" .
```

Exclude a directory:

```
grep -r --exclude-dir=node_modules "TODO" .
```

Useful Examples for System Administration

Find failed SSH logins:

```
grep "Failed password" /var/log/auth.log
```

Find users using Bash:

```
grep "/bin/bash" /etc/passwd
```

Find commented lines:

```
grep "^#" config.conf
```

Find empty lines:

```
grep "^$" file.txt
```

Find lines containing IP addresses:

```
grep -E "[0-9]{1,3}(\.[0-9]{1,3}){3}" logfile.txt
```

Exit Status

Exit Code	Meaning
`0`	Match found
`1`	No match found
`2`	Error occurred

Example:

```
grep "hello" file.txt
```

```
echo $?
```

Advantages

- Very fast text searching.
- Supports powerful regular expressions.
- Can search one or many files.
- Integrates well with pipes (`|`).
- Essential for log analysis, scripting, and system administration.

Common Mistakes

Searching a directory without recursive mode:

```
grep "hello" mydir
```

Output:

```
grep: mydir: Is a directory
```

Correct:

```
grep -r "hello" mydir
```

Searching for special regex characters literally:

```
grep "." file.txt
```

This matches **every line**, because `.` is a regular expression meaning "any single character".

To search for a literal dot:

```
grep -F "." file.txt
```

or

```
grep "\." file.txt
```

Summary

`grep` is one of the most important Linux text-processing tools. It searches files or command output for matching text or regular expressions and is widely used for log analysis, configuration searches, troubleshooting, scripting, and filtering command output. Mastering options like `-i`, `-v`, `-n`, `-r`, `-w`, `-E`, and `-F` will cover the majority of real-world `grep` usage.

2. `egrep` (Extended Global Regular Expression Print)

`egrep` is a Linux/Unix command used to **search for lines in a file that match an Extended Regular Expression (ERE)**.

It is equivalent to:

```
grep -E
```

> **Note:** On modern Linux systems, `egrep` is considered deprecated. You should use `grep -E` instead, as it provides the same functionality.

Syntax

```
egrep [OPTIONS] PATTERN [FILE...]
```

or

```
grep -E [OPTIONS] PATTERN [FILE...]
```

Why use `egrep`?

Unlike basic `grep`, `egrep` supports **extended regular expressions** without requiring backslashes for special characters.

For example:

- `|`
- `+`
- `?`
- `()`
- `{}`

work directly.

Examples

1. Search for a word

```
egrep "error" logfile.txt
```

Output:

```
Disk error detected
```

---

2. Search multiple words using OR (|)

```
egrep "error|warning" logfile.txt
```

Matches:

```
Disk error detected
Memory warning
```

Equivalent to:

```
grep -E "error|warning" logfile.txt
```

---

3. Match one or more characters (+)

```
echo "hellooo" | egrep "o+"
```

Output:

```
hellooo
```

+ means **one or more** occurrences.

---

4. Match optional character (?)

```
echo "color" | egrep "colou?r"
```

Matches:

```
color
```

Also matches:

```
colour
```

---

5. Group patterns using parentheses

```
egrep "(cat|dog)s?" animals.txt
```

Matches:

```
cat
cats
dog
dogs
```

---

6. Match repeated characters ({})

Exactly 3 digits:

```
echo "123" | egrep "^[0-9]{3}$"
```

Matches:

```
123
```

Does not match:

```
12
1234
```

---

7. Search recursively

```
egrep -r "main|printf" .
```

Searches all files in the current directory and subdirectories.

8. Ignore case

```
egrep -i "linux" file.txt
```

Matches:

```
Linux
LINUX
linux
LiNuX
```

---

9. Show line numbers

```
egrep -n "error" logfile.txt
```

Output:

```
25:Disk error detected
```

---

10. Count matching lines

```
egrep -c "error" logfile.txt
```

Output:

```
8
```

---

11. Invert the match

```
egrep -v "error" logfile.txt
```

Displays all lines **except** those containing "error".

12. Search multiple files

```
egrep "root|admin" file1.txt file2.txt
```

Output:

```
file1.txt:root
file2.txt:admin
```

Common Options

Option	Description
`-i`	Ignore case
`-v`	Invert the match
`-n`	Show line numbers
`-c`	Count matching lines
`-l`	Show only filenames with matches
`-r` or `-R`	Recursive search
`-o`	Show only the matching text
`-w`	Match whole words only
`-x`	Match the entire line
`-A n`	Show `n` lines after a match
`-B n`	Show `n` lines before a match
`-C n`	Show `n` lines before and after a match

`grep` vs `egrep`

Feature	`grep`	`egrep` (`grep -E`)
Regular expressions	Basic (BRE)	Extended (ERE)
`	` (OR)	|
+	\+	+
?	\?	?
()	\(\)	()
{}	\{ \}	{}

Example:

Basic `grep`:

```
grep "cat\|dog" file.txt
```

Extended `egrep`:

```
egrep "cat|dog" file.txt
```

Practical Examples

Find `.jpg` or `.png` files

```
ls | egrep "\.jpg$|\.png$"
```

Find users whose names start with `a` or `b`

```
cut -d: -f1 /etc/passwd | egrep "^[ab]"
```

Find IP addresses (simple pattern)

```
egrep "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+" access.log
```

Find email addresses (simple pattern)

```
egrep "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}" users.txt
```

Exit Status

Exit Code	Meaning
`0`	At least one match found
`1`	No matches found
`2`	An error occurred (e.g., invalid regular expression or missing file)

Best Practice

Although `egrep` still works on many systems, the recommended modern usage is:

```
grep -E "pattern" file.txt
```

instead of:

```
egrep "pattern" file.txt
```

This is more portable and aligns with current GNU `grep` recommendations.

Summary

- **Purpose:** Search text using **extended regular expressions (ERE)**.
- **Modern equivalent:** `grep -E`
- **Best for:** Complex pattern matching using `|`, `+`, `?`, `()`, and `{}` without escaping them.
- **Common uses:** Searching logs, filtering text, finding patterns in files, validating formats (such as emails or IP addresses), and recursively searching source code.

3.fgrep

`fgrep` is a Linux/Unix command used to **search for fixed text (literal strings)** in files. Unlike `grep`, it **does not interpret regular expressions (regex)**, making it useful when you want to search for text that contains special characters such as `.`, `*`, `[`, or `?`.

> **Note:** On most modern Linux systems, `fgrep` is equivalent to:
>
> `grep -F`

Syntax

```
fgrep [OPTIONS] "string" file
```

or

```
grep -F [OPTIONS] "string" file
```

How it works

- Searches for an **exact string**
- Does **not** treat regex characters specially
- Faster than regular `grep` for literal text searches
- Can search one or multiple files

Examples

1. Search for a word

```
fgrep "Linux" file.txt
```

Output

```
I love Linux
Linux is powerful
```

2. Search multiple files

```
fgrep "error" log1.txt log2.txt
```

Output

```
log1.txt:error opening file
log2.txt:error reading configuration
```

---

3. Ignore case

```
fgrep -i "linux" file.txt
```

Matches

```
Linux
LINUX
linux
LiNuX
```

---

4. Show line numbers

```
fgrep -n "main" program.c
```

Output

```
15:int main()
```

---

5. Count matching lines

```
fgrep -c "error" logfile
```

Output

```
7
```

Meaning there are **7 matching lines**.

---

6. Invert the match

Show lines **not** containing "error".

```
fgrep -v "error" logfile
```

---

7. Search recursively

```
fgrep -r "TODO" project/
```

Searches every file under the `project` directory.

---

8. Search text containing regex characters

Suppose a file contains

```
Version 1.2.3
```

Using

```
fgrep "1.2.3" file.txt
```

finds the exact text.

With regular `grep`, the dots (`.`) are regex wildcards unless escaped.

---

9. Search for multiple fixed strings

Create a file:

```
patterns.txt
```

Contents:

```
Linux
Ubuntu
Fedora
```

Run:

```
fgrep -f patterns.txt file.txt
```

It finds any line containing **Linux**, **Ubuntu**, or **Fedora**.

Common options

Option	Description
`-i`	Ignore case
`-n`	Show line numbers
`-c`	Count matching lines
`-v`	Show non-matching lines
`-l`	Print only filenames containing matches
`-L`	Print filenames with no matches
`-r`	Search recursively
`-w`	Match whole words only
`-x`	Match the entire line
`-f file`	Read search strings from a file
`-o`	Print only the matched text

Difference between `grep`, `fgrep`, and `egrep`

Command	Meaning	Supports Regex?	Equivalent
`grep`	General search	Yes	`Grep`
`fgrep`	Fixed string search	No	`grep -F`
`egrep`	Extended regex search	Yes (extended)	`grep -E`

Modern Linux systems recommend using:

```
grep -F
```

instead of:

```
fgrep
```

and

```
grep -E
```

instead of:

```
egrep
```

because `fgrep` and `egrep` are considered legacy command names.

Exit status

Exit Code	Meaning
0	Match found
1	No match found
2	An error occurred (e.g., file not found)

Example:

```
fgrep "Linux" file.txt
```

```
echo $?
```

Output:

```
0
```

Practical use cases

- Searching log files for exact error messages
- Finding configuration keys in configuration files
- Searching source code for exact function names
- Finding IP addresses, URLs, or version numbers
- Searching for text containing regex metacharacters without escaping them

---

Summary

- `fgrep` searches for **literal (fixed) strings**.
- It **does not interpret regular expressions**.
- It is equivalent to `grep -F`.
- It is generally **faster than `grep`** for exact string searches.
- On modern systems, **`grep -F` is the preferred form** over `fgrep`.

4 . awk

`awk` is a powerful Linux/Unix command-line utility used for **searching, processing, filtering, and formatting text**, especially text organized into **rows and columns**. It is one of the most useful tools for working with log files, CSV files, configuration files, and command output.

The name **AWK** comes from its creators:

- **A**lfred Aho
- **W**eter Weinberger
- **K**ernighan (Brian Kernighan)

Syntax

```
awk 'pattern { action }' file
```

or

```
command | awk 'pattern { action }'
```

- **pattern** – Specifies which lines to process.
- **action** – Specifies what to do with matching lines.
- **file** – The input file.

If no pattern is given, `awk` processes every line.

How awk Works

For each input line:

1. Reads one line.
2. Splits it into fields.
3. Tests the pattern.
4. Executes the action.
5. Repeats until the file ends.

Fields and Variables

By default:

- Fields are separated by spaces or tabs.
- Each column becomes a field.

Example file:

```
John 25 Engineer
Alice 30 Doctor
Bob 28 Teacher
```

Field variables:

- `$1` → First column
- `$2` → Second column
- `$3` → Third column
- `$0` → Entire line

Common Built-in Variables

Variable	Meaning
`$0`	Entire current line
`$1`	First field
`$2`	Second field
`$NF`	Last field
`NF`	Number of fields
`NR`	Current line number
`FNR`	Line number within current file
`FS`	Input field separator
`OFS`	Output field separator
`RS`	Record separator
`ORS`	Output record separator

Examples

1. Print entire file

```
awk '{print}' file.txt
```

Output:

```
John 25 Engineer
Alice 30 Doctor
```

---

2. Print first column

```
awk '{print $1}' file.txt
```

Output

```
John
Alice
```

---

3. Print first and third columns

```
awk '{print $1, $3}' file.txt
```

Output

```
John Engineer
Alice Doctor
```

---

4. Print line numbers

```
awk '{print NR, $0}' file.txt
```

Output

```
1 John 25 Engineer
2 Alice 30 Doctor
```

5. Print last column

```
awk '{print $NF}' file.txt
```

Output

```
Engineer
Doctor
Teacher
```

---

6. Print only lines matching a pattern

```
awk '/Alice/' file.txt
```

Output

```
Alice 30 Doctor
```

---

7. Print rows where age is greater than 28

```
awk '$2 > 28' file.txt
```

Output

```
Alice 30 Doctor
```

---

8. Print multiple conditions

```
awk '$2 >= 25 && $2 <= 28'
```

---

9. Sum a column

Suppose:

```
10
20
30
40
```

Command

```
awk '{sum += $1} END {print sum}' numbers.txt
```

Output

```
100
```

---

10. Average

```
awk '{sum += $1} END {print sum/NR}' numbers.txt
```

Output

```
25
```

---

11. Count lines

```
awk 'END {print NR}' file.txt
```

---

12. Count matching lines

```
awk '/error/ {count++} END {print count}' logfile
```

---

13. Specify a different field separator

CSV example:

```
John,25,Engineer
Alice,30,Doctor
```

Command

```
awk -F',' '{print $1,$3}' employees.csv
```

Output :

```
John Engineer
Alice Doctor
```

14. Change output separator

```
awk 'BEGIN{OFS="-"} {print $1,$2,$3}'
```

Output

```
John-25-Engineer
```

---

15. BEGIN and END blocks

```
awk '
BEGIN {
    print "Start"
}
{
    print $1
}
END {
    print "Finished"
}
' file.txt
```

Output

```
Start
John
Alice
Bob
Finished
```

---

16. Use with `ps`

```
ps -ef | awk '{print $1,$2,$8}'
```

Shows:

User

PID

Command

---

17. Use with `df`

```
df -h | awk '{print $1,$5}'
```

Displays:

- Filesystem
- Usage percentage

---

18. Use with `ls`

```
ls -l | awk '{print $9,$5}'
```

Displays:

- Filename
- File size

---

19. Print every second line

```
awk 'NR % 2 == 0'
```

---

20. Print lines between two patterns

```
awk '/START/,/END/' file.txt
```

Conditional Statements

```
awk '
{
    if ($2 > 30)
        print $1
}
' file.txt
```

Loops

For loop

```
awk '
{
    for(i=1;i<=NF;i++)
        print $i
}
' file.txt
```

While loop

```
awk '
{
    i=1
    while(i<=NF){
        print $i
        i++
    }
}
'
```

User-defined Variables

```
awk -v limit=50 '$2 > limit'
```

Built-in Functions

Function	Description
`length()`	Length of a string
`substr()`	Extract substring
`index()`	Find substring
`split()`	Split string into an array
`tolower()`	Convert to lowercase
`toupper()`	Convert to uppercase
`gsub()`	Replace all occurrences
`sub()`	Replace first occurrence
`sprintf()`	Format text
`system()`	Execute a shell command

Example:

```
awk '{print toupper($1)}'
```

Practical Use Cases

- Filter log files
- Process CSV or TSV files
- Generate reports
- Extract columns from command output
- Count words or lines
- Sum or average numeric values
- Search and replace text
- Parse configuration files
- Monitor system information

Advantages

- Very fast for text processing.
- Built into almost every Linux/Unix system.
- Combines filtering, formatting, and calculations in one tool.
- Works seamlessly in shell pipelines.
- Excellent for processing structured text.

Limitations

- Syntax can be difficult for beginners.
- Not ideal for complex data structures or object-oriented programming.
- Less suitable for handling very large binary data.

Common Options

Option	Description
`-F fs`	Set the input field separator
`-f file`	Read the AWK program from a file
`-v var=value`	Assign a variable before execution
`--version`	Display the AWK version

Example:

```
awk --version
```

Summary

`awk` is one of the most powerful text-processing tools in Linux. It reads input line by line, splits each line into fields, and performs actions based on patterns. It's widely used by system administrators, developers, and DevOps engineers for processing logs, generating reports, extracting data, and automating text manipulation tasks. Mastering `awk` can significantly improve your efficiency when working with structured text on the command line.

5. sed

`sed` (**S**tream **Ed**itor) is a powerful Linux/Unix command-line utility used to **search, filter, edit, and transform text**. It processes text line by line without opening it in an interactive editor.

It is commonly used in shell scripts and for automating text modifications.

Syntax

```
sed [OPTIONS] 'COMMAND' file
```

or

```
command | sed 'COMMAND'
```

What sed Can Do

- Search for text
- Replace text
- Delete lines
- Insert new lines
- Append text
- Print selected lines
- Modify configuration files
- Process logs
- Perform bulk text editing

Basic Examples

1. Display a file

```
sed '' file.txt
```

Example

```
sed '' names.txt
```

Output

```
Alice
Bob
Charlie
```

Although `cat` is normally used for this purpose, `sed` can also print file contents.

---

2. Replace text

```
sed 's/old/new/' file.txt
```

Example

File:

```
I like Linux.
```

Command

```
sed 's/Linux/Ubuntu/' file.txt
```

Output

```
I like Ubuntu.
```

Only the **first occurrence** on each line is replaced.

---

3. Replace every occurrence

Use the g (global) flag.

```
sed 's/apple/orange/g' fruits.txt
```

Input

```
apple apple banana apple
```

Output

```
orange orange banana orange
```

4. Edit the original file

Normally, sed only prints the modified output.

To save changes directly:

```
sed -i 's/old/new/g' file.txt
```

Example

```
sed -i 's/http/https/g' config.txt
```

The file is modified permanently.

---

5. Create a backup before editing

```
sed -i.bak 's/foo/bar/g' file.txt
```

This creates

```
file.txt
file.txt.bak
```

---

6. Delete a line

Delete line 3:

```
sed '3d' file.txt
```

Delete lines 2 to 5:

```
sed '2,5d' file.txt
```

Delete blank lines:

```
sed '/^$/d' file.txt
```

---

7. Print only specific lines

Print line 5:

```
sed -n '5p' file.txt
```

Print lines 10 to 20:

```
sed -n '10,20p' file.txt
```

---

8. Delete lines matching a pattern

```
sed '/error/d' logfile
```

Input

```
OK
error
Running
error
Done
```

Output

```
OK
Running
Done
```

---

9. Print lines matching a pattern

```
sed -n '/error/p' logfile
```

Output

```
error
error
```

---

10. Insert a line before another line

```
sed '/Linux/i\Welcome' file.txt
```

Input

```
Linux
Windows
```

Output

```
Welcome
Linux
Windows
```

---

11. Append a line

```
sed '/Linux/a\Ubuntu' file.txt
```

Output

```
Linux
Ubuntu
Windows
```

---

12. Replace an entire line

```
sed '/Linux/c\Operating System' file.txt
```

Output

```
Operating System
Windows
```

---

13. Remove leading spaces

```
sed 's/^ *//'
```

Input

```
     Hello
```

Output

```
Hello
```

---

14. Remove trailing spaces

```
sed 's/ *$//'
```

---

15. Remove both leading and trailing spaces

```
sed 's/^ *//;s/ *$//'
```

---

16. Number lines

```
sed = file.txt
```

or

```
sed '=' file.txt
```

---

17. Print every second line

```
sed -n '2~2p' file.txt
```

Output :

```
Line2
Line4
Line6
```

---

18. Replace using regular expressions

Replace all numbers with NUMBER

```
sed 's/[0-9]\+/NUMBER/g'
```

Input

```
Order 123
Item 456
```

Output

```
Order NUMBER
Item NUMBER
```

---

19. Read from a pipe

```
echo "Linux" | sed 's/Linux/Ubuntu/'
```

Output

```
Ubuntu
```

---

20. Multiple commands

```
sed -e 's/foo/bar/' -e 's/hello/hi/'
```

Useful Options

Option	Description
`-n`	Suppress automatic printing
`-i`	Edit file in place
`-e`	Execute multiple commands
`-f file`	Read commands from a script file
`-r` or `-E`	Use extended regular expressions (GNU `sed` commonly supports `-E`)
`-z`	Treat input as NUL-separated instead of newline-separated (GNU `sed`)

Common Regular Expressions

Pattern	Meaning
`^`	Start of line
`$`	End of line
`.`	Any single character
`*`	Zero or more of the previous character
`.*`	Any number of any characters
`[abc]`	`a`, `b`, or `c`
`[^abc]`	Any character except `a`, `b`, or `c`
`[0-9]`	Any digit
`[A-Z]`	Uppercase letter
`[a-z]`	Lowercase letter

Real-World Examples

Change all IP addresses

```
sed 's/192\.168\.1\.10/10.0.0.10/g' config.txt
```

Comment every line

```
sed 's/^/# /' file.txt
```

Uncomment lines

```
sed 's/^#//'
```

Delete comments

```
sed '/^#/d' file.txt
```

Replace tabs with spaces

```
sed 's/\t/    /g'
```

Convert Windows line endings to Unix

```
sed 's/\r$//' file.txt
```

Exit Status

Exit Code	Meaning
`0`	Success
Non-zero	An error occurred (such as an invalid command or file access issue)

Advantages

- Very fast for stream processing.
- Doesn't require opening files in an editor.
- Ideal for shell scripting and automation.
- Handles large files efficiently.

Limitations

- Complex editing commands can be difficult to read.
- Not intended for interactive editing.
- Some features differ between GNU `sed` (common on Linux) and BSD `sed` (common on macOS), especially `-i` behavior and certain extensions.

`sed` vs Similar Commands

Command	Purpose
`sed`	Stream editing and text transformation
`grep`	Search for matching text
`awk`	Pattern scanning, text processing, and reporting
`tr`	Translate or delete individual characters
`cut`	Extract specific columns or fields
`sort`	Sort lines of text
`perl`	Advanced text processing with rich scripting capabilities

Summary

`sed` is one of the most versatile text-processing tools in Linux. It excels at automated editing tasks such as substitutions, deletions, insertions, and line filtering, making it an essential utility for shell scripting, log processing, and configuration file management.

6.diff

The `diff` command in Linux and Unix is used to **compare two files line by line** and display the differences between them. It is commonly used by programmers, system administrators, and anyone who needs to compare text files.

Syntax

```
diff [OPTIONS] FILE1 FILE2
```

- `FILE1` – First file to compare.
- `FILE2` – Second file to compare.

Basic Example

file1.txt

```
Apple
Banana
Orange
Mango
```

file2.txt

```
Apple
Banana
Grapes
Mango
```

Run:

```
diff file1.txt file2.txt
```

Output:

```
3c3
< Orange
---
> Grapes
```

Meaning

- `3c3`
    - Line **3** in file1 changed (`c`) to line **3** in file2.
- `<` indicates content from the first file.
- `>` indicates content from the second file.

Common Options

`-u` (Unified Format)

Shows differences in a format commonly used for patches and version control.

```
diff -u file1.txt file2.txt
```

Example output:

```
--- file1.txt
+++ file2.txt
@@ -1,4 +1,4 @@
 Apple
 Banana
-Orange
+Grapes
 Mango
```

This is the most commonly used format.

`-y` (Side-by-Side Comparison)

Displays both files side by side.

```
diff -y file1.txt file2.txt
```

Example:

```
Apple           Apple
Banana          Banana
Orange        | Grapes
Mango           Mango
```

`indicates the lines differ.`

`--suppress-common-lines`

Use with `-y` to show only differing lines.

```
diff -y --suppress-common-lines file1.txt file2.txt
```

Output:

```
Orange       | Grapes
```

`-q` (Brief Output)

Only reports whether files differ.

```
diff -q file1.txt file2.txt
```

Output:

```
Files file1.txt and file2.txt differ
```

If identical:

```
Files file1.txt and file2.txt are identical
```

`-i` (Ignore Case)

```
diff -i file1.txt file2.txt
```

Example:

```
HELLO
```

and

```
hello
```

are considered the same.

`-w` (Ignore All Whitespace)

```
diff -w file1.txt file2.txt
```

Useful when differences are only spaces or tabs.

`-b` (Ignore Changes in Amount of Whitespace)

```
diff -b file1.txt file2.txt
```

Treats multiple spaces as a single space.

`-B` (Ignore Blank Lines)

```
diff -B file1.txt file2.txt
```

Blank lines are ignored during comparison.

`-r` (Compare Directories Recursively)

```
diff -r dir1 dir2
```

Compares all files and subdirectories.

Example output:

```
Only in dir2: notes.txt
Files dir1/config.conf and dir2/config.conf differ
```

`-N`

Treat missing files as empty.

```
diff -rN old_project new_project
```

Useful for generating patches.

`-c` (Context Format)

Shows surrounding lines around changes.

```
diff -c file1.txt file2.txt
```

`--color`

Highlight differences with colours.

```
diff --color file1.txt file2.txt
```

Understanding diff Symbols

Symbol	Meaning
<	Line from the first file
>	Line from the second file
a	Add
d	Delete
c	Change

Examples:

```
5a6
```

Add line 6 from file2 after line 5 in file1.

```
4d3
```

Delete line 4 from file1.

```
7c7
```

Change line 7.

Comparing Directories

```
diff -r folder1 folder2
```

Example:

```
Only in folder2: test.txt
Files folder1/main.c and folder2/main.c differ
```

Exit Status

Exit Code	Meaning
0	Files are identical
1	Files differ
2	Error occurred (e.g., file not found)

Example:

```
diff file1 file2
```

```
echo $?
```

Output:

```
1
```

The files are different.

Common Use Cases

- Compare configuration files.
- Check changes before deployment.
- Compare source code versions.
- Generate patches (`diff -u`).
- Compare backup directories.
- Verify generated output against expected output.

Related Commands

Command	Purpose
`cmp`	Compare two files byte by byte.
`comm`	Compare two sorted files line by line.
`sdiff`	Interactive side-by-side comparison.
`patch`	Apply changes generated by `diff`.
`git diff`	Compare changes in Git repositories.

Summary

- **Purpose:** Compare files or directories and show differences.
- **Best for:** Source code, configuration files, text documents, and directory comparisons.
- **Most useful options:**
  - `-u` – Unified diff (most common)
  - `-y` – Side-by-side comparison
  - `-q` – Brief comparison
  - `-r` – Compare directories recursively
  - `-w` – Ignore whitespace
  - `-i` – Ignore case
  - `-B` – Ignore blank lines
  - `--color` – Colourise differences

`diff` is one of the most important Linux text-processing utilities and is widely used in software development, system administration, and version control workflows.

7. cmp

The `cmp` command in Linux is used to **compare two files byte by byte**. It is commonly used to determine whether two files are identical or to find the first difference between them.

Syntax

```
cmp [OPTIONS] FILE1 FILE2
```

- `FILE1` – First file to compare.
- `FILE2` – Second file to compare.

How `cmp` Works

- If the files are **identical**, `cmp` produces **no output** and returns exit status `0`.
- If the files are **different**, it reports the **first byte and line** where they differ.
- If one file ends before the other, it reports that **EOF (End Of File)** was reached.

Basic Examples

1. Compare two files

```
cmp file1.txt file2.txt
```

Example output:

```
file1.txt file2.txt differ: byte 25, line 3
```

This means the first difference occurs at:

- Byte: **25**
- Line: **3**

---

2. Compare identical files

```
cmp original.txt backup.txt
```

Output:

```
(no output)
```

Check the exit status:

```
echo $?
```

Output:

```
0
```

This means the files are identical.

---

3. Compare binary files

```
cmp image1.png image2.png
```

Output:

```
image1.png image2.png differ: byte 14567, line 1
```

Since binary files may not contain newline characters, the line number is often `1`.

Common Options

`-l`

Display **every differing byte**.

```
cmp -l file1.bin file2.bin
```

Example output:

```
25 141 142
38 150 151
```

Columns mean:

1. Byte position
2. Byte value in first file (octal)
3. Byte value in second file (octal)

`-s`

Silent mode.

```
cmp -s file1.txt file2.txt
```

No output is displayed.

Check the result:

```
echo $?
```

- `0` → Files are identical
- `1` → Files differ
- `2` → Error occurred

Useful in shell scripts.

Example:

```
if cmp -s file1 file2; then
    echo "Files are identical"
else
    echo "Files are different"
fi
```

`-b`

Show differing bytes as characters.

```
cmp -b file1 file2
```

Example:

```
file1 file2 differ: byte 15, line 2 is 141 a 142 b
```

`-i SKIP`

Skip bytes before comparison.

```
cmp -i 100 file1 file2
```

Skip the first 100 bytes in both files.

Skip different amounts:

```
cmp -i 100:200 file1 file2
```

- Skip 100 bytes in `file1`
- Skip 200 bytes in `file2`

`-n LIMIT`

Compare only the first specified number of bytes.

```
cmp -n 1024 file1 file2
```

Compares only the first 1024 bytes.

Exit Status

Exit Code	Meaning
`0`	Files are identical
`1`	Files differ
`2`	Error (missing file, permission denied, etc.)

`cmp` vs `diff`

Feature	`cmp`	`diff`
Comparison type	Byte-by-byte	Line-by-line
Best for	Binary files	Text files
Reports	First difference (or all with `-l`)	Detailed text changes
Identical files	No output	No output (default)
Script friendly	Yes	Yes

Practical Examples

Verify copied file

```
cp report.pdf report_copy.pdf
cmp report.pdf report_copy.pdf
```

If nothing is printed, the copy is identical.

Compare ISO images

```
cmp ubuntu.iso backup.iso
```

Compare firmware binaries

```
cmp firmware_v1.bin firmware_v2.bin
```

Use in a shell script

```
#!/bin/bash

if cmp -s config.old config.new; then
    echo "No changes detected."
else
    echo "Configuration has changed."
fi
```

Advantages

- Very fast for checking file equality.
- Works well with both text and binary files.
- Can identify the exact byte where files first differ.
- Useful in automation and shell scripts.

Limitations

- Does not show detailed text differences like `diff`.
- By default, stops at the first difference.
- Less useful than `diff` when comparing source code or documents.

Summary

- **Purpose:** Compare two files byte by byte.
- **Best for:** Verifying identical files, comparing binary files, and scripting.
- **Basic command:**

  ```
  cmp file1 file2
  ```

- **Useful options:**
  - `-s` – Silent comparison
  - `-l` – Show all differing bytes
  - `-b` – Display differing bytes as characters
  - `-i` – Skip bytes before comparing
  - `-n` – Compare only a specified number of bytes