Introduction to Operating Systems

6. Text Processing

2026-11-03

USO Team

Bibliography

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Sections 2.2.2 - 2.2.3 - `cat`, *Standard Input and Standard Output*
     - Section 2.5 - *Intermediate Commands*
     - Section 2.12 - *Text Editors*
     - Section 2.14 - *Shell Input and Output*
   - Chapter 11 - *Introduction to Shell Scripts*
     - Sections 11.4 - 11.5 - *Exit Codes, Conditionals*
     - Section 11.10 - *Important Shell Script Utilities* (`awk`, `sed`)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 2 - *Utilizarea sistemului de fișiere*
     - Section 2.4 - *Redirectarea intrării sau ieșirii*
   - Chapter 4 - *Procese*
     - Sections 4.5.3 - 4.5.4 - *Înlănțuirea comenzilor, Comunicarea prin pipe-uri*
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Sections 6.6.1 - 6.6.3 - *Scrierea codului sursă, Editoare în mod text, Editoare în interfață grafică*
   - Chapter 7 - *Interfața în linia de comandă*
     - Sections 7.4 - 7.5 - *Expresii regulate, Prelucrare de text de bază*
   - Chapter 13 - *Automatizarea sarcinilor*
     - Section 13.3.2 - *Controlul fluxului*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Doug McIlroy (the inventor of the Unix pipe)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Text Processing

- Pipelines
- Line Filters
  - `cat`, `tac`, `head`, `tail`, `sort`, `uniq`, `nl`
- Search
  - regular expressions, `grep`
- Change Text
  - `cut`, `tr`, `sed`, `rev`
- Compute
  - `wc`, `uniq -c`, `awk`
- A Pipeline, Step by Step
- Exit Codes
- Text Editors
  - `nano`, `vim`, `emacs`, `helix`
- Graphical Editors
  - VS Code / VSCodium, Zed

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Pipelines

Chain the commands together

Bibliography

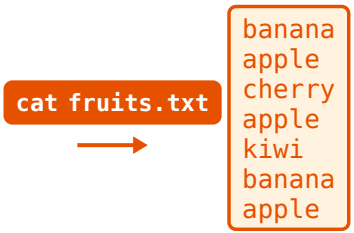
For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.14 - *Shell Input and Output*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Sections 4.5.3 - 4.5.4 - *Înlănțuirea comenzilor, Comunicarea prin pipe-uri*
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.5 - *Prelucrare de text de bază: filtre de text și one linere*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Pipeline

The text flows through the pipes, every command changes it



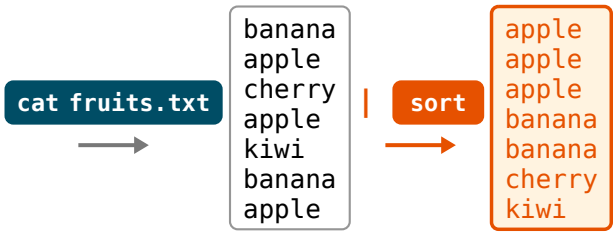

```
$ cat fruits.txt
```

💡 `cat` writes the text of `fruits.txt`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Pipeline

The text flows through the pipes, every command changes it



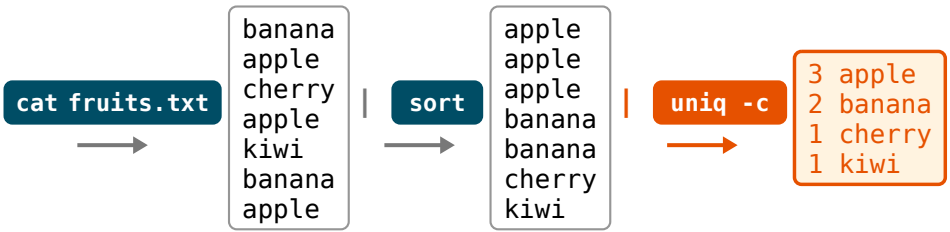

```
$ cat fruits.txt | sort
```

💡 `sort` reads it from the pipe: the same lines, in order

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Pipeline

The text flows through the pipes, every command changes it



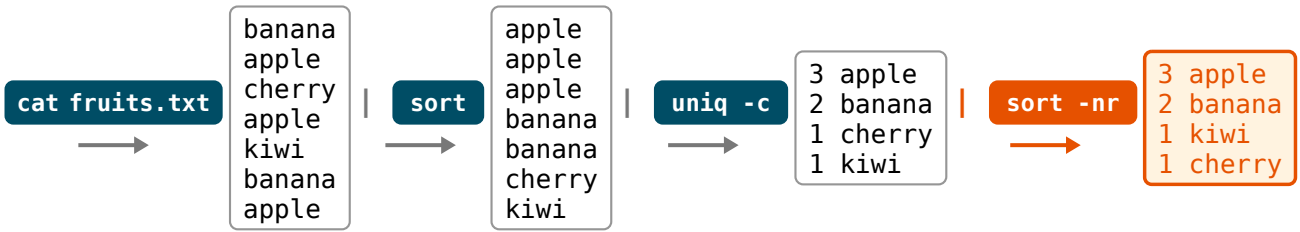

```
$ cat fruits.txt | sort | uniq -c
```

💡 `uniq -c` - every line once, with how many times it showed up (the terminal shows spaces before the count)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Pipeline

The text flows through the pipes, every command changes it




```
$ cat fruits.txt | sort | uniq -c | sort -nr
```

💡 `sort -nr` - the most frequent first: every command does **one** small thing, the pipeline does the job

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

A Typical Pipeline

A preview: which addresses asked for missing pages, and how many times?

Line Filters

which lines, in which order

`cat` `head` `tail`

`sort` `uniq` `nl` ...

```
$ tail -n 1000 access.log \
  | grep ' 404$' \
  | cut -d' ' -f1 \
  | sort | uniq -c
```



stdout

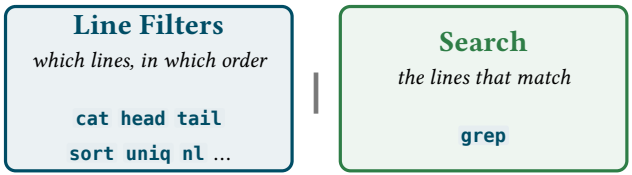
```
10.0.0.7 GET /index.html 200
10.0.0.9 GET /logo.png 404
10.0.0.7 GET /about.html 200
10.0.0.3 GET /old.html 404
10.0.0.9 GET /favicon.ico 404
10.0.0.5 GET /index.html 200
10.0.0.9 GET /admin 404
10.0.0.3 GET /old.html 404
```

💡 `tail -n 1000` - only the last 1000 lines of the log, see *Line Filters*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

A Typical Pipeline

A preview: which addresses asked for missing pages, and how many times?




```
$ tail -n 1000 access.log \
  | grep ' 404$' \
  | cut -d' ' -f1 \
  | sort | uniq -c
```

→

stdout

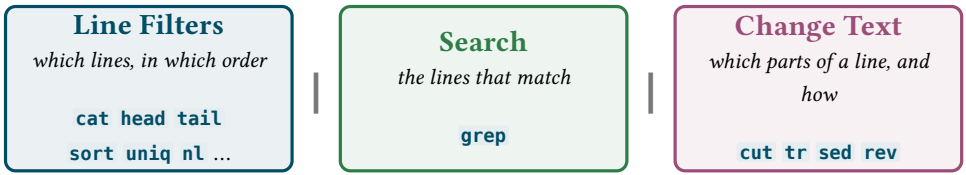
```
10.0.0.9 GET /logo.png 404
10.0.0.3 GET /old.html 404
10.0.0.9 GET /favicon.ico 404
10.0.0.9 GET /admin 404
10.0.0.3 GET /old.html 404
```

💡 `grep ' 404$'` - the lines that end with `404`, *page not found*, see *Search*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

A Typical Pipeline

A preview: which addresses asked for missing pages, and how many times?




```
$ tail -n 1000 access.log \
  | grep ' 404$' \
  | cut -d' ' -f1 \
  | sort | uniq -c
```

stdout

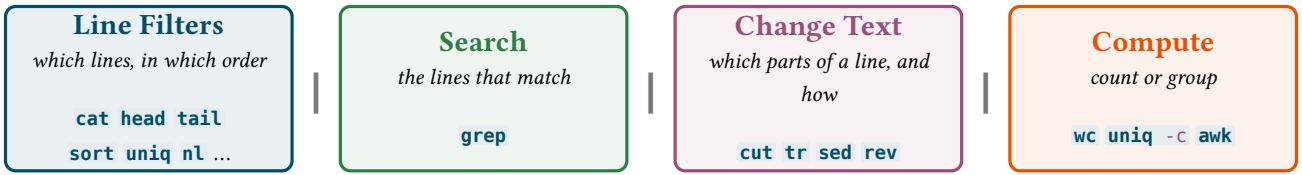
```
10.0.0.9
10.0.0.3
10.0.0.9
10.0.0.9
10.0.0.3
```

💡 `cut -d' ' -f1` - keep only the first field, the address, see *Change Text*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

A Typical Pipeline

A preview: which addresses asked for missing pages, and how many times?




```
$ tail -n 1000 access.log \
  | grep ' 404$' \
  | cut -d' ' -f1 \
  | sort | uniq -c
```

stdout

```
      2 10.0.0.3
      3 10.0.0.9
```

💡 `sort | uniq -c` - how many times every address shows up, see *Compute*; by the end, you can read every piece

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Line Filters

Work with whole lines: `cat`, `tac`, `head`, `tail`, `sort`, `uniq`, `nl`

Bibliography

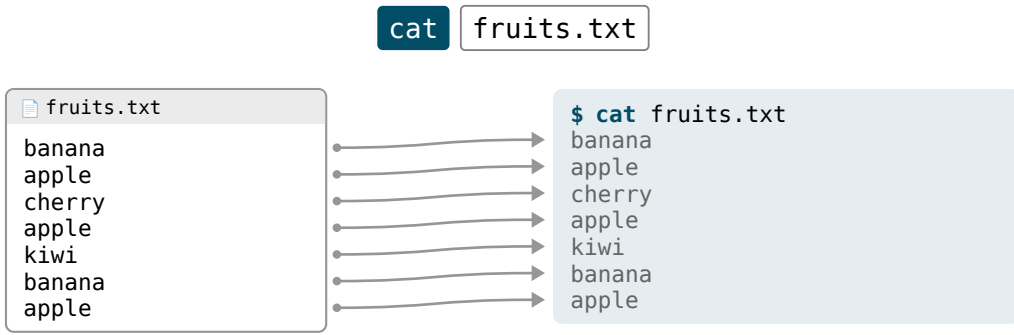
For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.2.2 - `cat`
     - Section 2.5 - *Intermediate Commands* (`head`, `tail`, `sort`)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.5 - *Prelucrare de text de bază: filtre de text și one linere*
3. the manual: `man nl`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

cat

🐱 Concatenate and display files

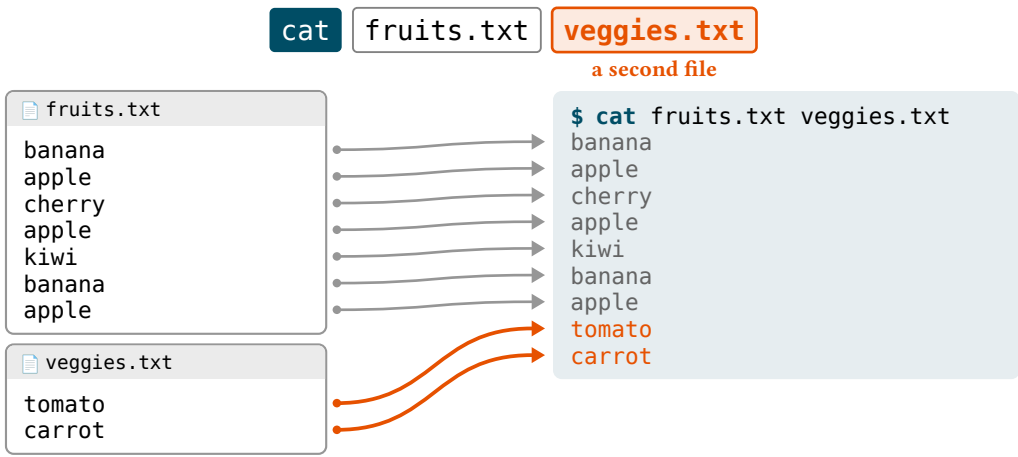


💡 display the contents of a file: every line, as it is, in the same order

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

cat

🐱 Concatenate and display files

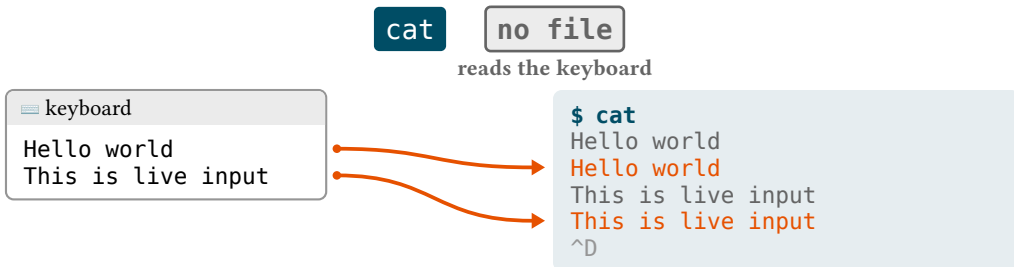


💡 several files, one after the other: *concatenate* them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`cat`

🐱 Concatenate and display files

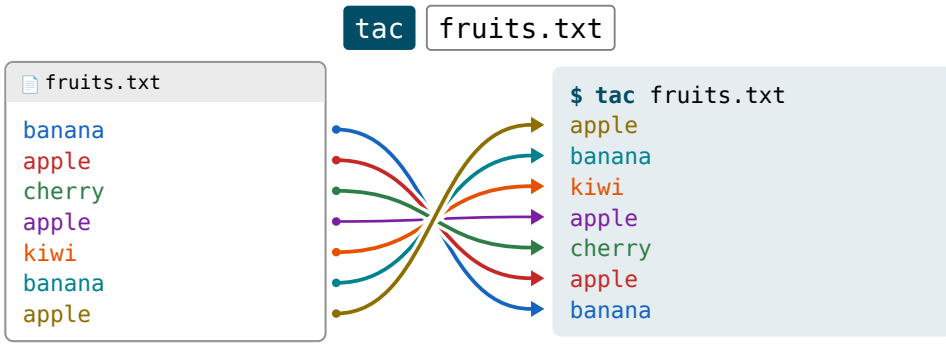


💡 no file: reads `stdin`, every line you type is printed back (orange), until `Ctrl` + `D`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

tac

🔃 `cat` backwards: the last line first

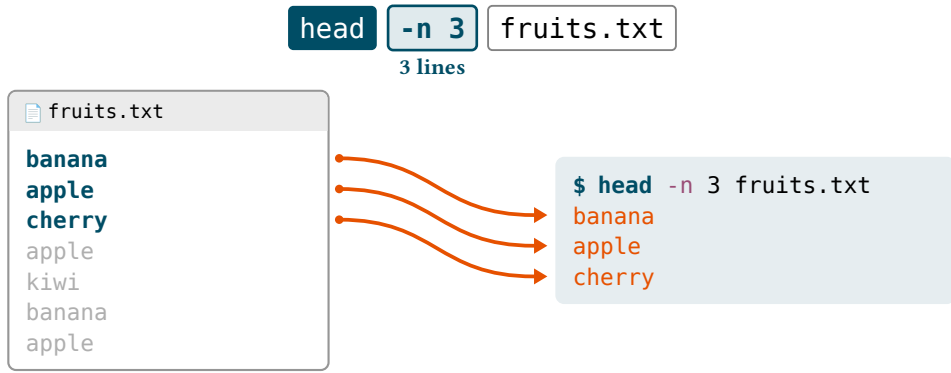


💡 the same lines, in the **reverse order**; every line stays as it is (`rev` reverses the characters, see *Change Text*)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`head` and `tail`

🔝 The first and 🔚 the last lines

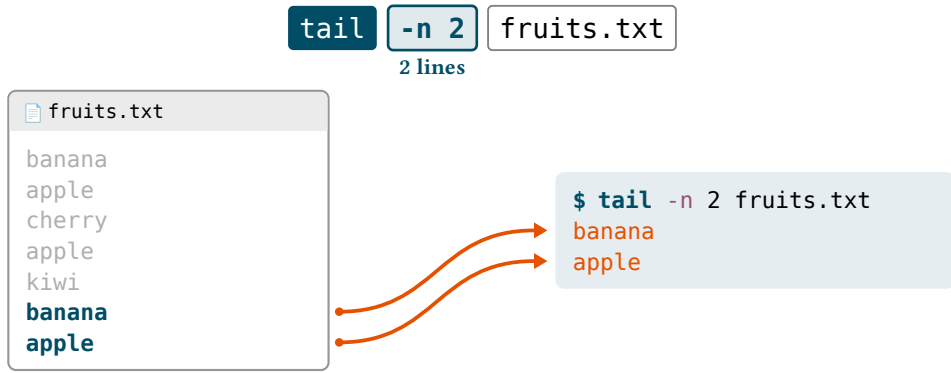


💡 the **first** lines; `-n` - how many, without it **10**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`head` and `tail`

🔝 The first and 🔚 the last lines

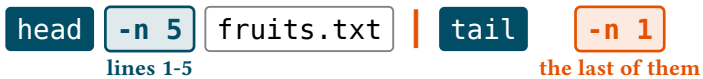


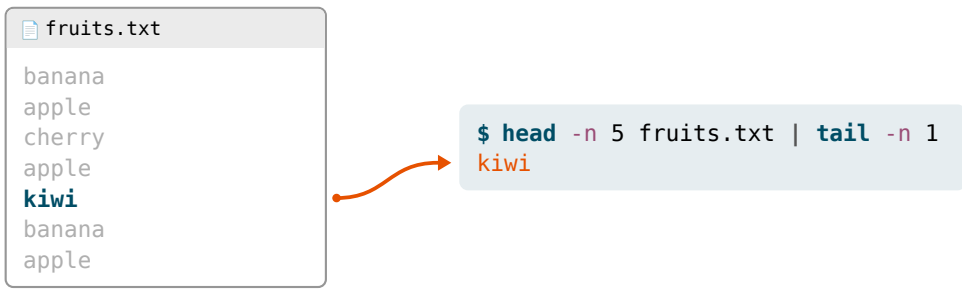
💡 the **last** lines; `-n` - how many, without it **10**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`head` and `tail`

🔝 The first and 🔚 the last lines





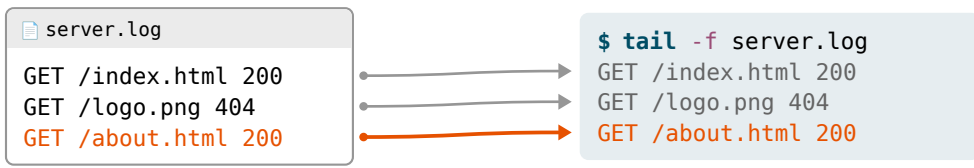
💡 combine them to extract a line from the middle: the **5th** line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`head` and `tail`

🔝 The first and 🔚 the last lines





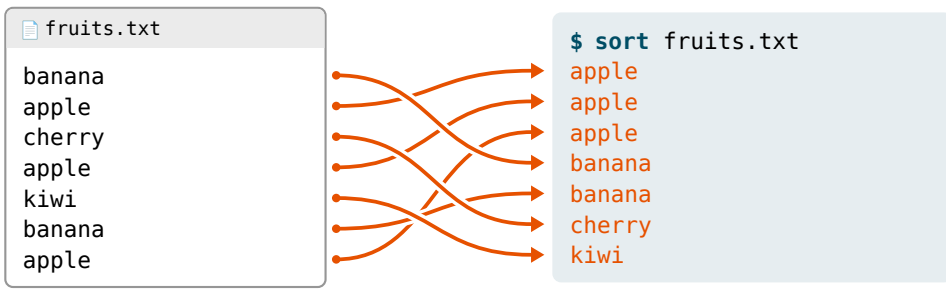
💡 `tail -f` keeps the file open and prints the new lines as they are written, until `Ctrl` + `C`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sort

🔤 Sort lines alphabetically or numerically

`sort` `fruits.txt`

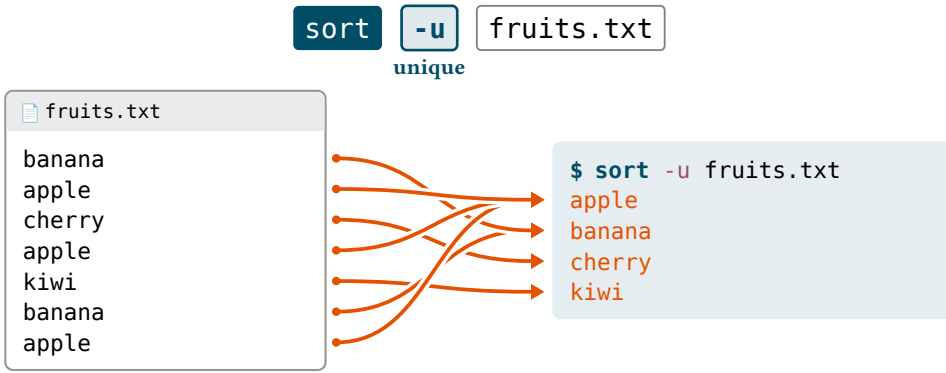


💡 in alphabetical order; `sort -r` - in *reverse* order

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sort

🔤 Sort lines alphabetically or numerically



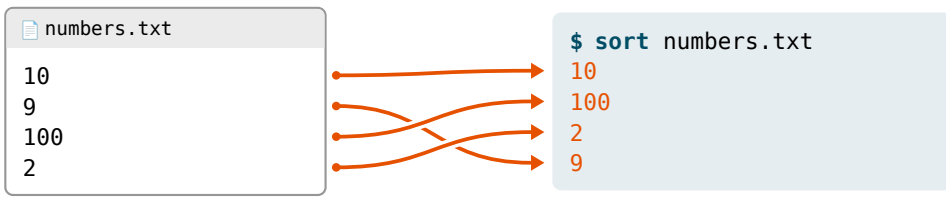
💡 `-u` - every line only once (*unique*): the same lines become one

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sort

🔤 Sort lines alphabetically or numerically

```
sort numbers.txt
```



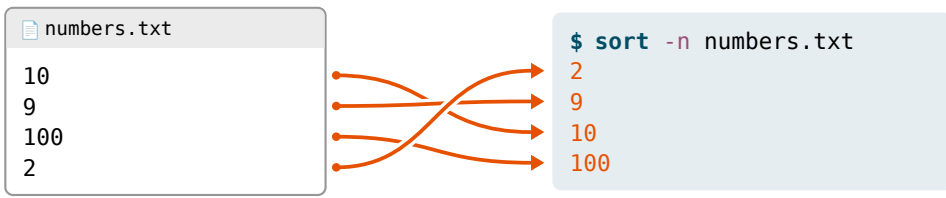
⚠️ numbers are sorted as text, character by character: `"100"` comes before `"2"`, as `1` < `2`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sort

🔤 Sort lines alphabetically or numerically



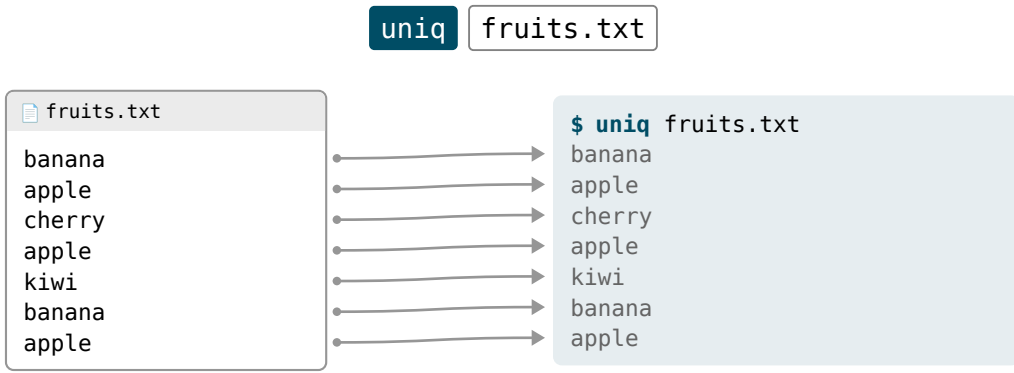


💡 `-n` - by the value of the number; `sort -t, -k2 -n` sorts by the **2nd field**, fields separated by `,`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

uniq

🔁 Filter out **adjacent** repeated lines



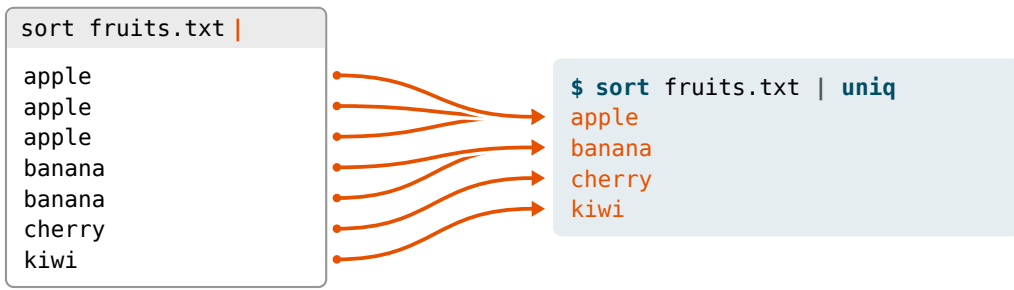
⚠️ `uniq` only compares a line with the **previous** one: no two `apple` lines are next to each other, nothing is removed

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`uniq`

🔁 Filter out **adjacent** repeated lines

`sort` `fruits.txt` | `uniq`



💡 `sort` first, so the identical lines are next to each other

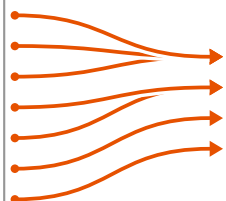
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

uniq

🔁 Filter out **adjacent** repeated lines

```
sort fruits.txt | uniq -c
                  count
```

```
sort fruits.txt |
apple
apple
apple
banana
banana
cherry
kiwi
```



```
$ sort fruits.txt | uniq -c
      3 apple
      2 banana
      1 cherry
      1 kiwi
```

💡 `-c` - how many times every line shows up

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

uniq

🔁 Filter out **adjacent** repeated lines

`sort` `fruits.txt` | `uniq` `-d`

only repeated

```
sort fruits.txt |
apple
apple
apple
banana
banana
cherry
kiwi
```

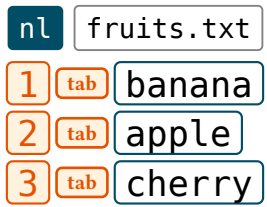
```
$ sort fruits.txt | uniq -d
apple
banana
```

💡 `-d` - only the lines that are repeated; `cherry` and `kiwi` show up once, they are dropped

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`nl`

🏷️ Number the lines



📄 fruits.txt

```
banana
apple
cherry
```

→

```
$ nl fruits.txt
     1  banana
     2  apple
     3  cherry
```

💡 *number lines*: a **new first column**, the number of the line, then a tab

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Search

Regular expressions and `grep`

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Jamie Zawinski (one of the authors of Netscape Navigator)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.5.1 - `grep`
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.4 - *Expresii regulate*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Regular Expression

A pattern that describes text

- most characters match **themselves**, a few are *special*: `.` is *any character*

the regex `a.p` in `fruits.txt`

```
banana
apple
cherry
apple
kiwi
banana
apple
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Regular Expression

A pattern that describes text

- most characters match **themselves**, a few are *special*: `.` is *any character*
- `grep 'regex' file` prints the lines that **match**

the regex `a.p` in `fruits.txt`

```
banana
apple
cherry
apple
kiwi
banana
apple
```

```
$ grep 'a.p' fruits.txt
apple
apple
apple
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Regular Expression

A pattern that describes text

- most characters match **themselves**, a few are *special*: `.` is *any character*
- `grep 'regex' file` prints the lines that **match**
- used by

`grep` *search* — `less` `vim` *search with* `/`

`sed` *replace*

`awk` *columns*

`Python` `JavaScript` `C` `Rust` ... *every programming language*

the regex `a.p` in `fruits.txt`

```
banana
apple
cherry
apple
kiwi
banana
apple
```

```
$ grep 'a.p' fruits.txt
apple
apple
apple
```


Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Regular Expression

A pattern that describes text

- most characters match **themselves**, a few are *special*: `.` is *any character*
- `grep 'regex' file` prints the lines that **match**
- used by

`grep`

search

`less` `vim`

search with `/`

the regex `a.p` in `fruits.txt`

banana
apple

🐚 put the regex between `' '`

the files in the current directory `banana.txt` `fruits.txt`

✅ `grep 'ba*' fruits.txt` ***grep*** *gets* `grep` `ba*` `fruits.txt`

❌ `grep ba* fruits.txt` ***grep*** *gets* `grep` `banana.txt` `fruits.txt`

without quotes, the **shell** replaces `ba*` (a glob) with the file names, before `grep` starts

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep 'cat' words.txt
```



💡 most characters match **themselves**: the text `cat`, anywhere in the line

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep 'c.l' words.txt
```



💡 `.` is **any single character**: `col`, `cal`

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep '^ca' words.txt
```



💡 `^` is the **start** of the line

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep 'r$' words.txt
```



💡 `$` is the **end** of the line

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep '[0-9]' words.txt
```

[0-9]

one digit

💡 `[ ]` is **one** of the characters inside (a *class*), `0-9` is a range

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep '^[^c]' words.txt
```



💡 `[^ ]` is one character that is **not** inside

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

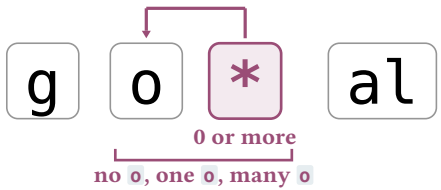
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep 'go*al' words.txt
```



💡 `*` repeats the **previous item**, zero or more times: `gal`, `goal`, `gooooal`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

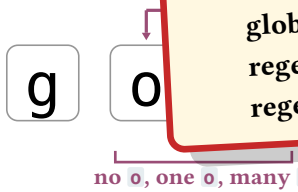
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep 'go*al' words.txt
```



⚠ `*` in a regex is not `*` in a glob!

glob (`ls`)	`*`	*any characters*
regex (`grep`)	`*`	**repeats** the previous item
regex (`grep`)	`.*`	*any characters*

```
banana
bat
orange
gal
goal
goooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

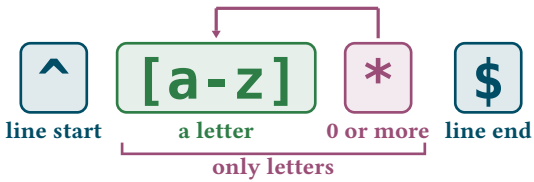
💡 `*` repeats the **previous item**, zero or more times: `gal`, `goal`, `gooooal`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep '^[a-z]*$' words.txt
```



💡 `*` after a class: any number of characters **from the class**, here the whole line is letters

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Basic Regular Expressions

Characters, anchors and classes - `grep 'regex'`

```
$ grep '\.' words.txt
```



a real dot

💡 `\` makes the next character **literal**: a real dot, `.` alone matches every line

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

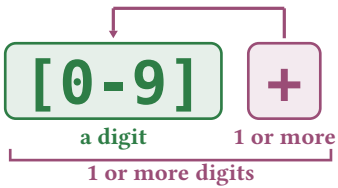
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E '[0-9]+' words.txt
```



💡 `+` repeats the previous item, **one or more** times, after a class: one or more digits

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

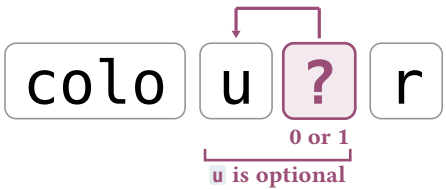
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E 'colou?r' words.txt
```



💡 `?` makes the previous item **optional**: zero or one time

`words.txt`

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

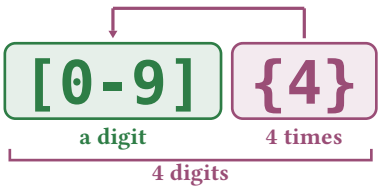
`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E '[0-9]{4}' words.txt
```



💡 `{n}` repeats the previous item **exactly** `n` times

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

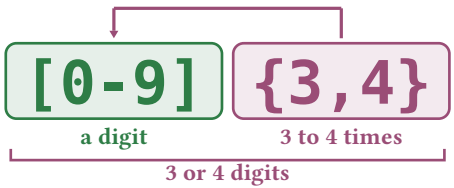
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E '[0-9]{3,4}' words.txt
```



💡 `{n,m}` repeats it **between** `n` and `m` times

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

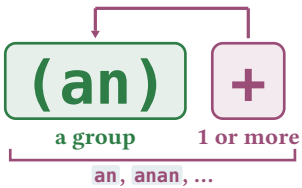
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E '(an)+' words.txt
```



💡 `( )` makes a **group**, `+` after it repeats the whole group: `an` in `orange`, `anan` in `banana`

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

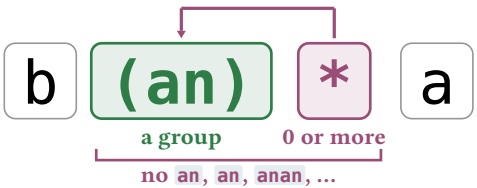
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E 'b(an)*a' words.txt
```



💡 `*` after a group: the whole group, zero or more times, `bat` has no `an` and still matches

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

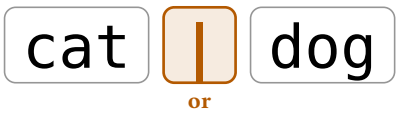
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E 'cat|dog' words.txt
```



💡 `|` is **or**: one of the alternatives

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Extended Regular Expressions

`grep -E 'regex'`, without `-E` they need a `\`: `\+`, `\?`, `\{ \}`, `\( \)`, `\|`

```
$ grep -E '\bcat\b' words.txt
```



💡 `\b` is a **word edge**: only the word `cat`, not `category`

`\+`, `\?`, `\|` (without `-E`) and `\b` are not POSIX, they are GNU `grep` extensions (Linux): on macOS, use `grep -E` and check `man grep`

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
goooooal
notes.txt
2026-10-04
call 0722 123 456
```

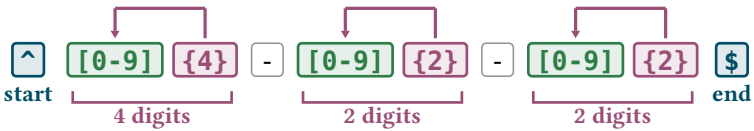
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

RegEx in Action

Put the pieces together, the matches are highlighted like `grep --color` does

```
$ grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2}$' words.txt
```



💡 a whole line that is a date

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

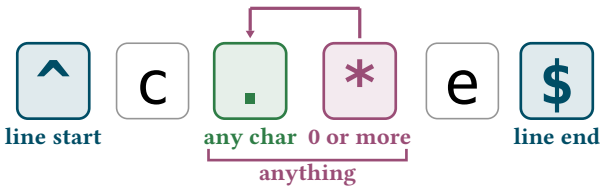
gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

RegEx in Action

Put the pieces together, the matches are highlighted like `grep --color` does

```
$ grep '^c.*e$' words.txt
```



💡 starts with `c`, **anything**, ends with `e`

📄 words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

gray lines do not match, `grep` does not print them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

RegEx in Action

Put the pieces together, the matches are highlighted like `grep --color` does

```
$ grep -E '\b[0-9]{3}\b' words.txt
```



💡 a **number** of exactly 3 digits: `0722` has 4

words.txt

```
color
colour
cat
category
concatenate
dog
banana
bat
orange
gal
goal
gooooal
notes.txt
2026-10-04
call 0722 123 456
```

*gray lines do not match, **`grep`** does not print them*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`grep` Options

Change what `grep` prints

Option	Prints
`-n`	the line **n**umbers too
`-c`	only the **c**ount of lines
`-v`	the lines that do **not** match
`-i`	*ignores* upper / lower case
`-o`	**o**nly the matching part
`-q`	nothing (*quiet*), see *exit codes*

```
$ grep -n 'an' fruits.txt
1:banana
6:banana
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`grep` Options

Change what `grep` prints

Option	Prints
`-n`	the line **n**umbers too
`-c`	only the **c**ount of lines
`-v`	the lines that do **not** match
`-i`	*ignores* upper / lower case
`-o`	**o**nly the matching part
`-q`	nothing (*quiet*), see *exit codes*

```
$ grep -n 'an' fruits.txt
1:banana
6:banana
$ grep -c 'apple' fruits.txt
3
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`grep` Options

Change what `grep` prints

Option	Prints
`-n`	the line **n**umbers too
`-c`	only the **c**ount of lines
`-v`	the lines that do **not** match
`-i`	*ignores* upper / lower case
`-o`	**o**nly the matching part
`-q`	nothing (*quiet*), see *exit codes*

```
$ grep -n 'an' fruits.txt
1:banana
6:banana
$ grep -c 'apple' fruits.txt
3
$ grep -v 'apple' fruits.txt
banana
cherry
kiwi
banana
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`grep` Options

Change what `grep` prints

Option	Prints
`-n`	the line **n**umbers too
`-c`	only the **c**ount of lines
`-v`	the lines that do **not** match
`-i`	*ignores* upper / lower case
`-o`	**o**nly the matching part
`-q`	nothing (*quiet*), see *exit codes*

```
$ grep -n 'an' fruits.txt
1:banana
6:banana
$ grep -c 'apple' fruits.txt
3
$ grep -v 'apple' fruits.txt
banana
cherry
kiwi
banana
$ grep -o 'an' fruits.txt
an
an
an
an
```

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`grep` Options

Change what `grep` prints

Option	Prints
`-n`	the line **n**umbers too
`-c`	only the **c**ount of lines
`-v`	the lines that do **not** match
`-i`	*ignores* upper / lower case
`-o`	**o**nly the matching part
`-q`	nothing (*quiet*), see *exit codes*

```
$ grep -n 'an' fruits.txt
1:banana
6:banana
$ grep -c 'apple' fruits.txt
3
$ grep -v 'apple' fruits.txt
banana
cherry
kiwi
banana
$ grep -o 'an' fruits.txt
an
an
an
an
$ grep -q 'kiwi' fruits.txt
```

💡 `-q` prints nothing: the answer is in the *exit code* of `grep` (see *Exit Codes*)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

RegEx Cheat Sheet

Everything else matches itself, `grep 'regex'` or `grep -E 'regex'`

Characters

`.`	any single character	`c.l`
`[abc]`	one of the characters	`c[ao]l`
`[0-9]`	one character in the range	`[a-z]`
`[^abc]`	one character **not** inside	`[^0-9]`

Repetitions - of the previous item

`*`	0 or more times	`go*al`
`+`	1 or more times `-E`	`[0-9]+`
`?`	0 or 1 time `-E`	`colou?r`
`{n,m}`	`n` to `m` times, `{n}` exactly `n` `-E`	`[0-9]{4}`

Anchors - match a place, not a character

`^`	the start of the line	`^ca`
`$`	the end of the line	`r$`
`\b`	a word edge	`\bcat\b`

Groups, or, escape

`( )`	a group, repeated as one item `-E`	`(an)+`
`|`	one of the alternatives (*or*) `-E`	`cat|dog`
`\`	the next character is literal	`\.`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Change Text

Which parts of a line, and how they change: `cut`, `tr`, `sed`, `rev`

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 11 - *Introduction to Shell Scripts*
     - Section 11.10 - *Important Shell Script Utilities* (`sed`)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.5 - *Prelucrare de text de bază: filtre de text și one linere*
3. the manuals: `man cut`, `man tr`, `man sed`, `man rev`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Column

A line split into **fields** by a **delimiter**



📄 students.csv

```
name,age,city
alice,30,Paris
bob,25,London
carol,22,Bucharest
```

💡 a **CSV** file (*comma-separated values*): every line is a row of a table, the `,` separates the columns

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Column

A line split into **fields** by a **delimiter**



access.log

```
10.0.0.7 GET /index.html 200
10.0.0.9 GET /logo.png 404
```

💡 a log file: a **space** separates the fields; `/etc/passwd` uses `:`, many programs a **tab**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Column

A line split into **fields** by a **delimiter**



people.txt

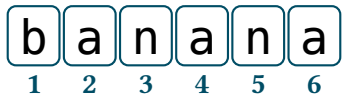
```
alice 30 Paris
bob   25 London
```

⚠ **every** space is a delimiter: two spaces next to each other have an **empty** field between them, `25` is field **4**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is a Column

A line split into **fields** by a **delimiter**



📄 fruits.txt

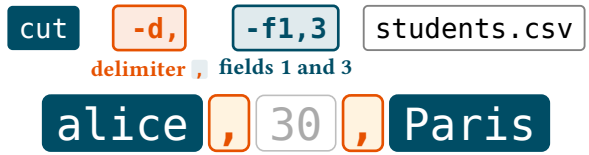
```
banana
apple
cherry
```

💡 no delimiter at all: the columns are the **positions** of the characters, from `1`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

cut

✂️ Keep some fields or some characters of every line



students.csv

```
name,age,city
alice,30,Paris
bob,25,London
carol,22,Bucharest
```

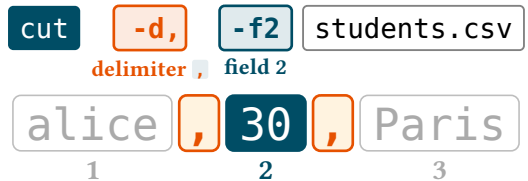
```
$ cut -d, -f1,3 students.csv
name,city
alice,Paris
bob,London
carol,Bucharest
```

💡 `-d` - how the line is split, `-f` - which fields to keep, printed with the same delimiter

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

cut

✂️ Keep some fields or some characters of every line



students.csv

```
name,age,city
alice,30,Paris
bob,25,London
carol,22,Bucharest
```

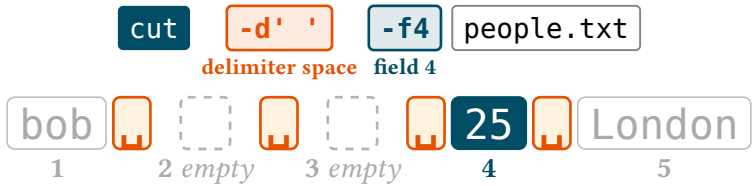
```
$ cut -d, -f2 students.csv
age
30
25
22
```

💡 `-f2-3` - the fields 2 to 3, `-f2-` - from field 2 to the end

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`cut`

✂️ Keep some fields or some characters of every line



people.txt

```
alice 30 Paris
bob   25 London
```

```
$ cut -d' ' -f4 people.txt

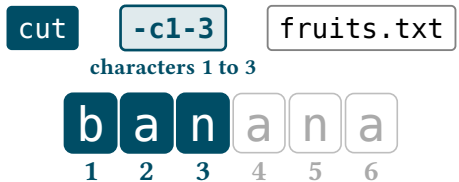
25
```

⚠️ `alice` has only 3 fields: an empty line; `tr` fixes this, next

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

cut

✂️ Keep some fields or some characters of every line



fruits.txt

```
banana
apple
cherry
```

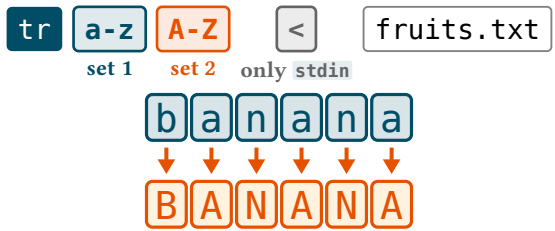
```
$ cut -c1-3 fruits.txt
ban
app
che
...
```

💡 `-c` counts the **characters**, there is no delimiter

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

tr

Translate, delete or squeeze characters



fruits.txt

```
banana
apple
cherry
```

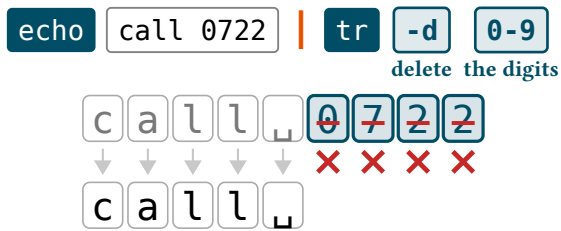
```
$ tr a-z A-Z < fruits.txt
BANANA
APPLE
CHERRY
```

set 1 → set 2, at the **same position**: `a` → `A`, `b` → `B`, ...; `tr` reads only `stdin`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

tr

🔡 Translate, delete or squeeze characters



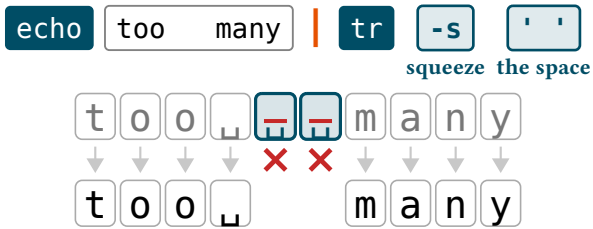

```
$ echo 'call 0722' | tr -d 0-9
call
```

💡 `-d` - one set: its characters are **deleted**, the others stay, the space too

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

tr

Translate, delete or squeeze characters



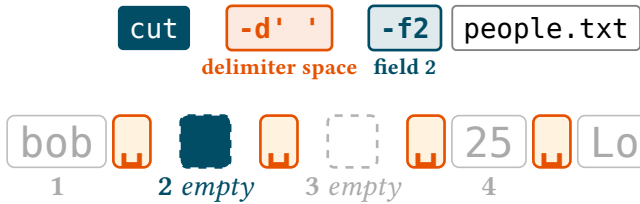

```
$ echo 'too   many' | tr -s ' '
too many
```

💡 `-s` - a **run** of the same character becomes only one: the first space stays, the others go

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Squeeze the Spaces, then `cut`

The example with more spaces, again



people.txt

```
alice 30 Paris
bob   25 London
```

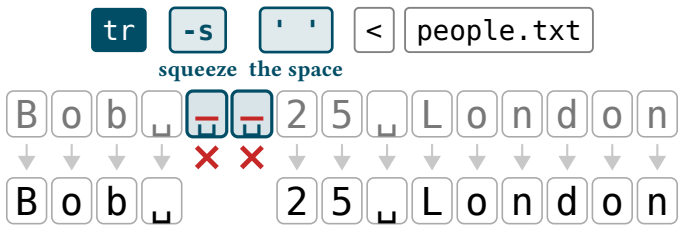
```
$ cut -d' ' -f2 people.txt
30
```

⚠️ for `bob`, field 2 is the **empty** field between the first two spaces

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Squeeze the Spaces, then `cut`

The example with more spaces, again



people.txt

```
alice 30 Paris
bob   25 London
```

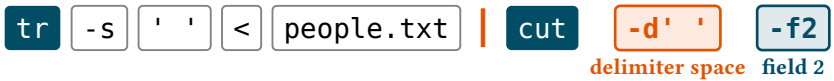
```
$ tr -s ' ' < people.txt
alice 30 Paris
bob 25 London
```

💡 squeeze the spaces: one space between the fields, no more empty fields

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Squeeze the Spaces, then `cut`

The example with more spaces, again



people.txt

```
alice 30 Paris
bob   25 London
```



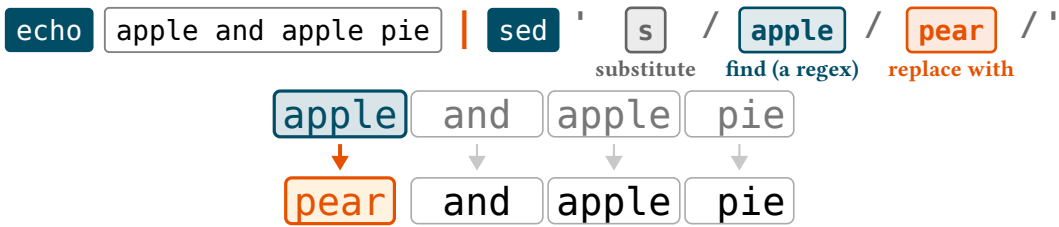
```
$ tr -s ' ' < people.txt | cut -d' ' -f2
30
25
```

✅ squeeze first, then `cut`: every line has the same fields

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sed

✏️ Substitute: replace the text that matches a regex



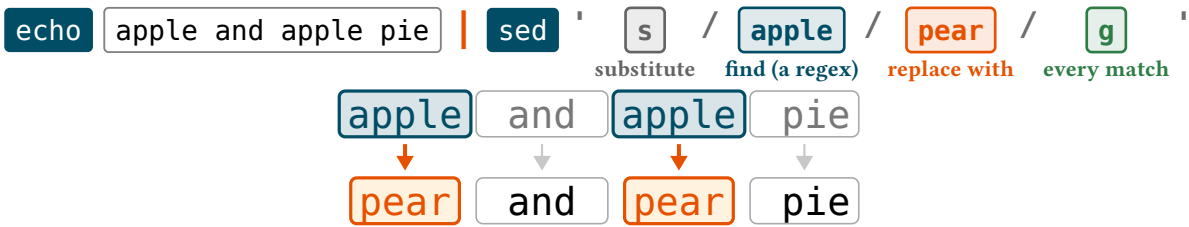

```
$ echo 'apple and apple pie' | sed 's/apple/pear/'
pear and apple pie
```

💡 only the **first** match of every line is replaced; `tr` changes characters, `sed` changes words

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sed

✏️ Substitute: replace the text that matches a regex



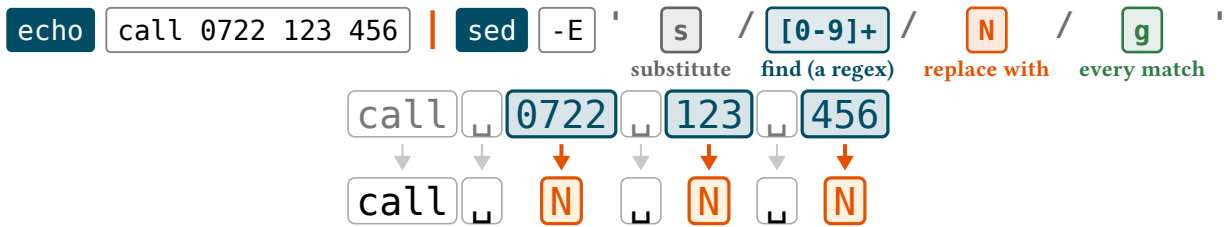

```
$ echo 'apple and apple pie' | sed 's/apple/pear/g'
pear and pear pie
```

💡 `g` - *global*: every match of the line is replaced

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sed

✏️ Substitute: replace the text that matches a regex



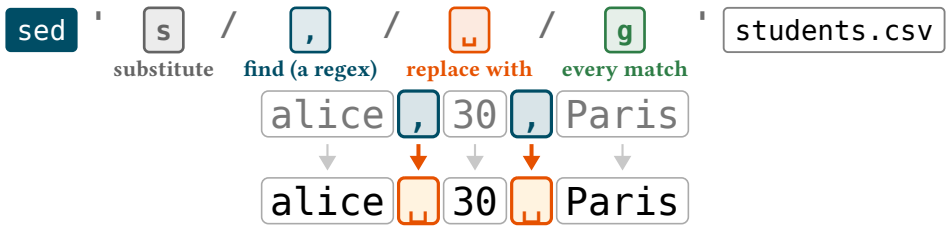

```
$ echo 'call 0722 123 456' | sed -E 's/[0-9]+/N/g'
call N N N
```

💡 the text to find is a **regex** (see *Search*), `-E` - extended, as `grep -E`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

sed

✏️ Substitute: replace the text that matches a regex



students.csv

```
name,age,city
alice,30,Paris
bob,25,London
```

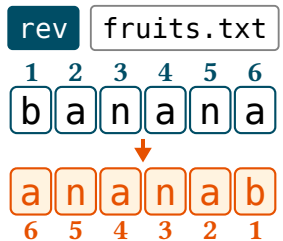
```
$ sed 's/,/ /g' students.csv
name age city
alice 30 Paris
bob 25 London
```

💡 change the delimiter of the columns: `,` becomes a space

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

rev

🔄 Reverse the characters of every line



📄 fruits.txt

```
banana
apple
cherry
```

```
$ rev fruits.txt
ananab
elppa
yrrehc
```

💡 every line is reversed, character by character; the **order of the lines** stays (that is `tac`)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Compute

Count and combine

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 11 - *Introduction to Shell Scripts*
     - Section 11.10 - *Important Shell Script Utilities* (`awk`)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.5 - *Prelucrare de text de bază: filtre de text și one linere*
3. the manuals: `man wc`, `man awk`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

🔢 Word count: lines, words and bytes

`wc` `fruits.txt`

7	7	44	fruits.txt
lines	words	bytes	the file

📄 fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

```
$ wc fruits.txt
 7  7 44 fruits.txt
```

💡 *word count*: the lines, the words, the bytes, then the name of the file

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

🔢 Word count: lines, words and bytes



📄 fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

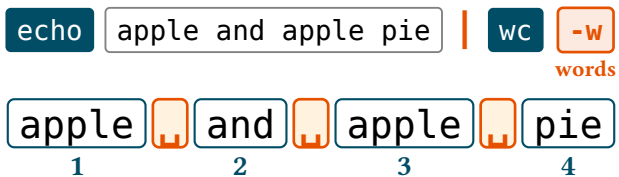
```
$ wc -l fruits.txt
7 fruits.txt
```

💡 `-l` - only the number of **lines**: 7 fruits

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

wc

🔢 Word count: lines, words and bytes

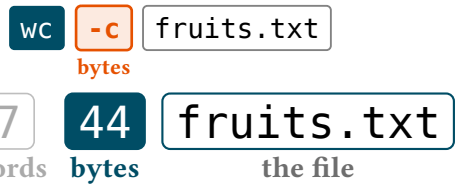



```
$ echo 'apple and apple pie' | wc -w
4
```

💡 `-w` - the **words**: groups of characters separated by spaces; from `stdin`, no file name

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

🔢 Word count: lines, words and bytes



fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

```
$ wc -c fruits.txt
44 fruits.txt
```

💡 `-c` - the **bytes**: 37 letters and 7 *newlines* (one at the end of every line), 44

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

🔢 Word count: lines, words and bytes



fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

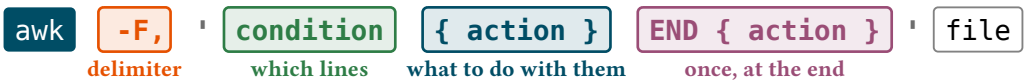
```
$ wc -l < fruits.txt
7
```

💡 the shell opens `fruits.txt`, `wc` reads `stdin`: it does not know the name of the file, it prints only the number

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is `awk`

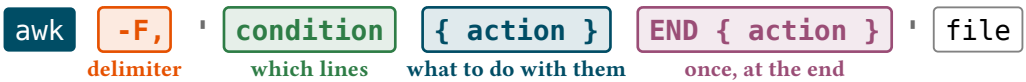
A small language for text split into fields



 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is `awk`

A small language for text split into fields





the whole line is `$0`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

What Is `awk`

A small language for text split into fields

```
awk -F, ' condition { action } END { action } ' file
```

`-F,` — delimiter; `condition` — which lines; `{ action }` — what to do with them; `END { action }` — once, at the end

```
alice , apple , 3
$1      $2      $3
```

the whole line is `$0`

- `awk` reads the lines one by one, splits each one into **fields**
- for every line: if the **condition** is true, run the **action**
- no condition: every line; no action: print the line

in `awk`	means
`$1`, `$2`, ...	the fields
`NF`	the number of fields
`NR`	the number of the line
`s`, `n`	variables: empty, `0` in a sum

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

awk

🧮 Compute with the fields of every line

```
awk -F, ' { s += $3 } END { print s } ' orders.csv
```

`-F,` — delimiter `,`

`{ s += $3 }` — every line: add field 3

`END { print s }` — at the end: print

3 + 1 + 2 + 5 + 1 + 2 = 14

📄 orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

→

📄 stdout

```
14
```

💡 `$3` is field 3, like `cut -f3`; the program is in `'...'`, so the shell does not change `$3`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

awk

🧮 Compute with the fields of every line

```
awk -F, ' NR > 1 { s += $3; n++ } END { print s / n } ' orders.csv
```

- `-F,` — delimiter `,`
- `NR > 1` — from line 2
- `{ s += $3; n++ }` — add, count
- `END { print s / n }` — at the end: the average

$$14 / 6 = 2.33333$$

📄 orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

→

📄 stdout

```
2.33333
```

💡 `NR` is the number of the line: skip the header; `n` counts the orders, 14 / 6 = 2.33

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

awk

🧮 Compute with the fields of every line

```
awk -F, '$2 == "apple" { s += $3 } END { print s }' orders.csv
```

- `-F,` — delimiter `,`
- `$2 == "apple"` — only the apples
- `{ s += $3 }` — add field 3
- `END { print s }` — at the end: print

3 + 2 + 1 + 2 = 8

📄 orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

→

📄 stdout

```
8
```

💡 a condition before `{ }`: only the lines where field 2 is `apple`; `awk` can search, cut and compute

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`awk` Conditions

Combine them, match a regex, use several of them

```
awk -F, '$2 == "apple" && $3 >= 2 { s += $3 } END { print s }'
```

- `-F,` — delimiter `,`
- `$2 == "apple" && $3 >= 2` — apples and at least 2
- `{ s += $3 }` — add field 3
- `END { print s }` — at the end



orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

→

stdout

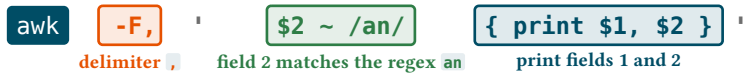
```
7
```

💡 `&&` - and, `||` - or, `!` - not, as in C; `dan` ordered only 1 apple, so he is skipped

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`awk` Conditions

Combine them, match a regex, use several of them





orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```



stdout

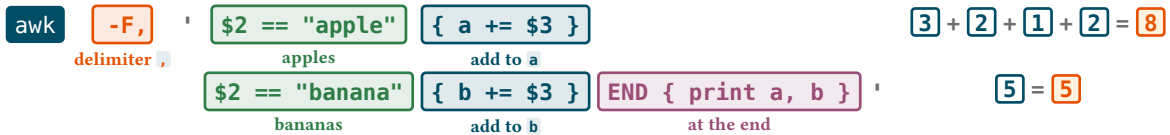
```
alice banana
```

💡 `~` - matches a regex (see *Search*), `!~` - does not match; `print $1, $2` puts a space between them

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`awk` Conditions

Combine them, match a regex, use several of them



orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

→

stdout

```
8 5
```

💡 every line is checked against **every** pair: two sums in one pass over the file

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

A Pipeline, Step by Step

How the text changes from one command to the next

Bibliography

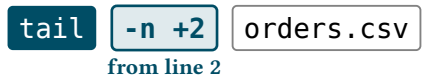
For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.14 - *Shell Input and Output*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.5.4 - *Comunicarea prin pipe-uri*
   - Chapter 7 - *Interfața în linia de comandă*
     - Section 7.5 - *Prelucrare de text de bază: filtre de text și one linere*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

How Many Customers Ordered Apples?

Every step adds one command to the pipeline



orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

stdout

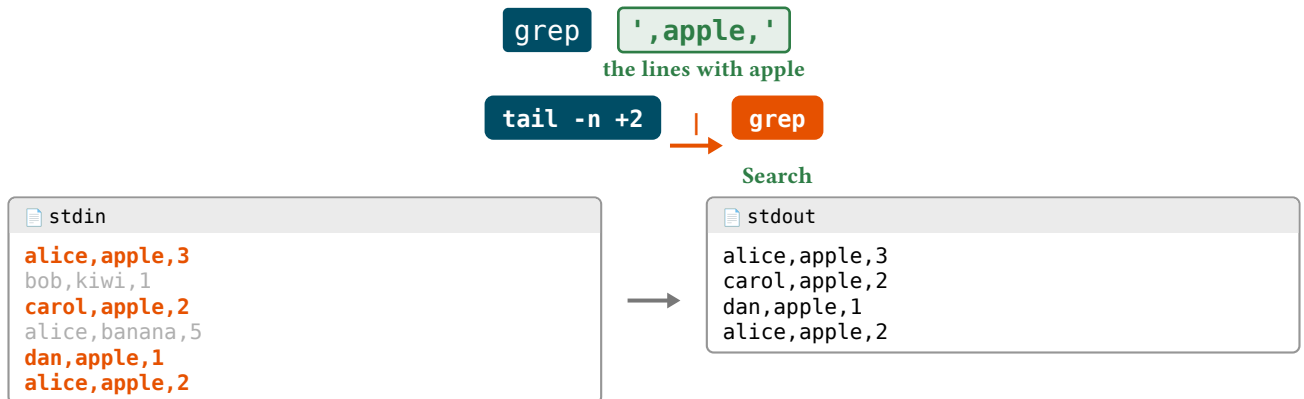
```
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

💡 the first line is the header (the names of the columns), not an order: skip it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

How Many Customers Ordered Apples?

Every step adds one command to the pipeline

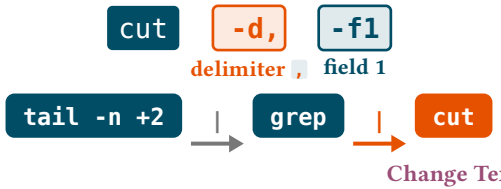


💡 keep only the orders of apples; the commas around `apple` skip a fruit like `pineapple`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

How Many Customers Ordered Apples?

Every step adds one command to the pipeline



stdin

```
alice,apple,3
carol,apple,2
dan,apple,1
alice,apple,2
```

stdout

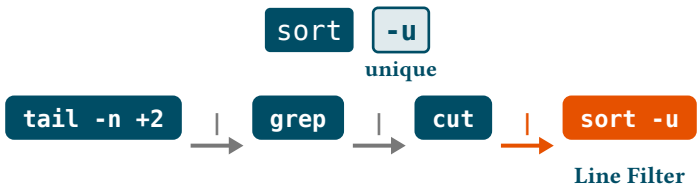
```
alice
carol
dan
alice
```

💡 keep only the name of the customer, the first field

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

How Many Customers Ordered Apples?

Every step adds one command to the pipeline

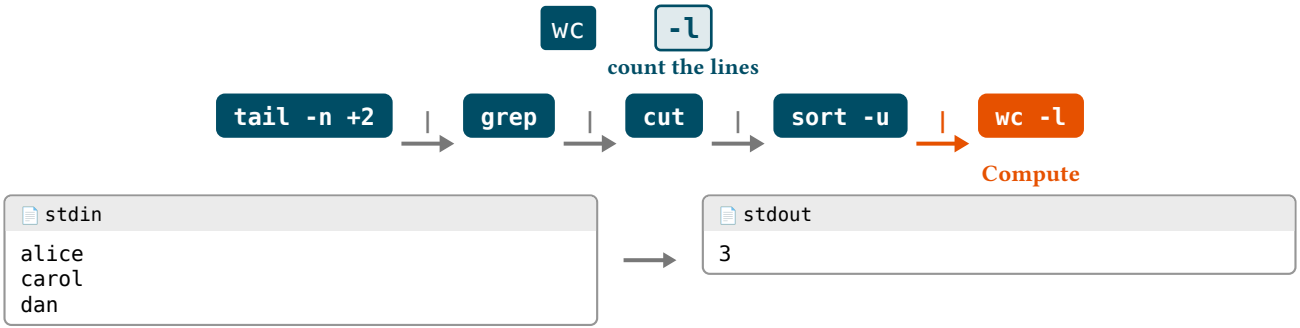


💡 `alice` ordered apples twice: sort the names and keep every name once

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

How Many Customers Ordered Apples?

Every step adds one command to the pipeline



✅ `tail -n +2 orders.csv | grep ',apple,' | cut -d, -f1 | sort -u | wc -l` → `3`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

How a command says if it worked

Bibliography

For this section

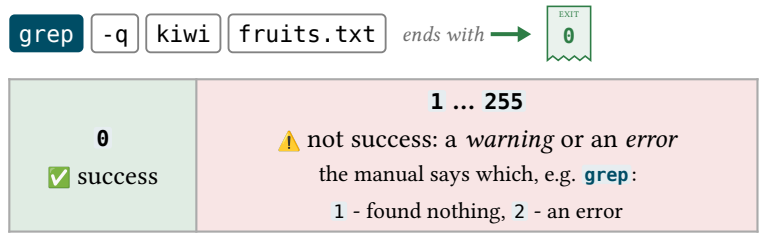
1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 11 - *Introduction to Shell Scripts*
     - Section 11.4 - *Exit Codes*
     - Section 11.5 - *Conditionals* (`&&` and `||`)
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 4 - *Procese*
     - Section 4.5.3 - *Înlănțuirea comenzilor*
   - Chapter 13 - *Automatizarea sarcinilor*
     - Section 13.3.2 - *Controlul fluxului*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



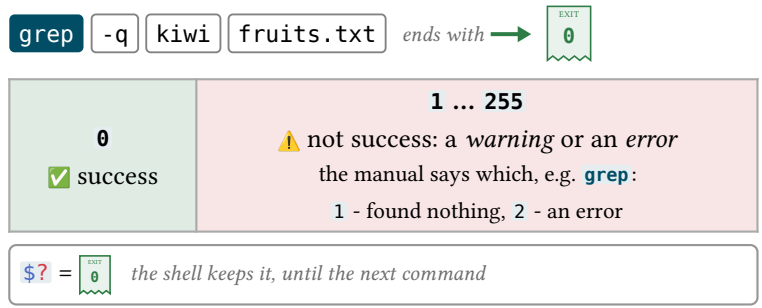
💡 every command ends with an *exit code*, `0` - `255`: `grep -q` found `kiwi`, `0`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



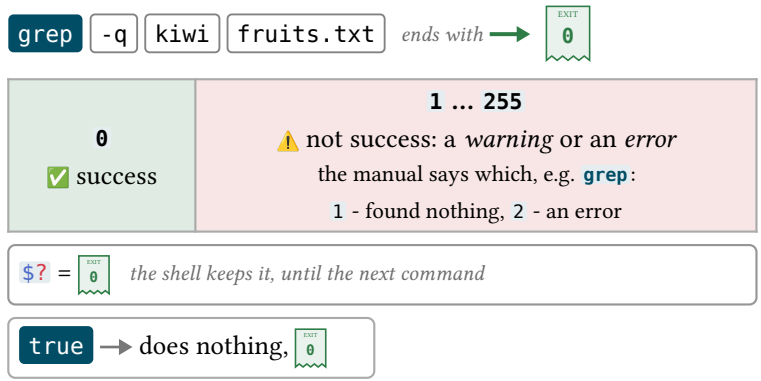
💡 `$?` - the exit code of the last command, it changes after **every** command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



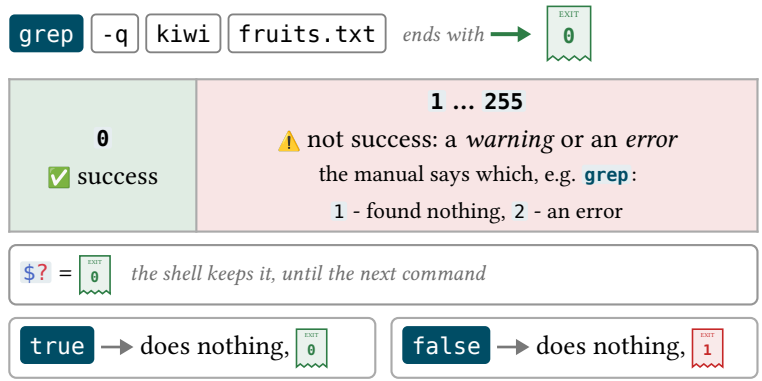
💡 `true` - a command that does nothing, it only ends with `0`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



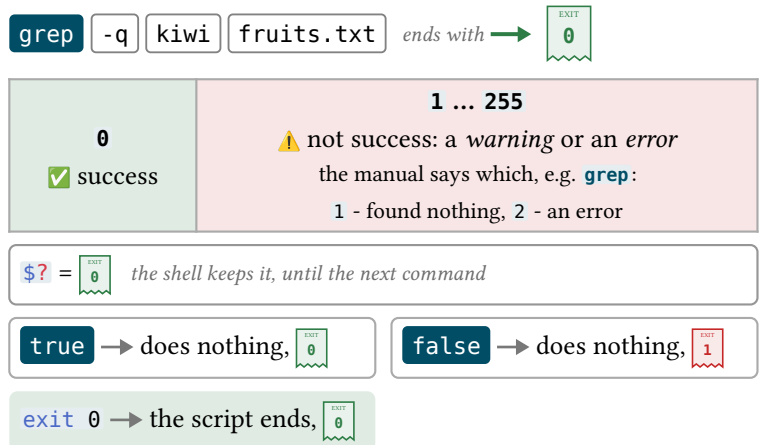
💡 `false` - a command that does nothing, it only ends with `1`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



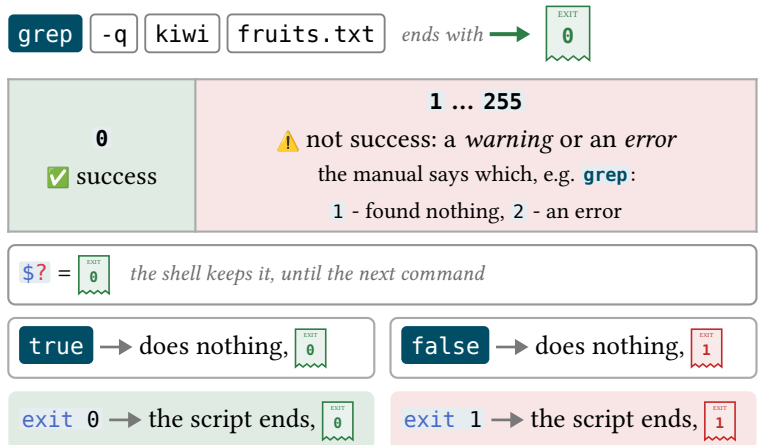
💡 `exit 0` - end the script with *success*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



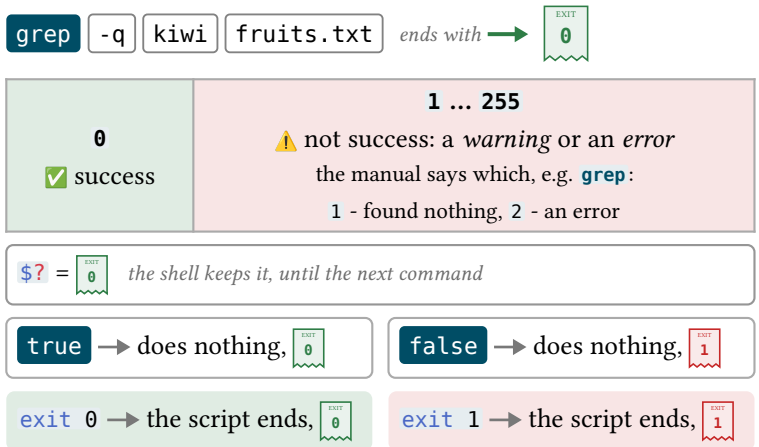
💡 `exit 1` (or `2`, `3`, ...) - end the script with *not success*, a warning or an error

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



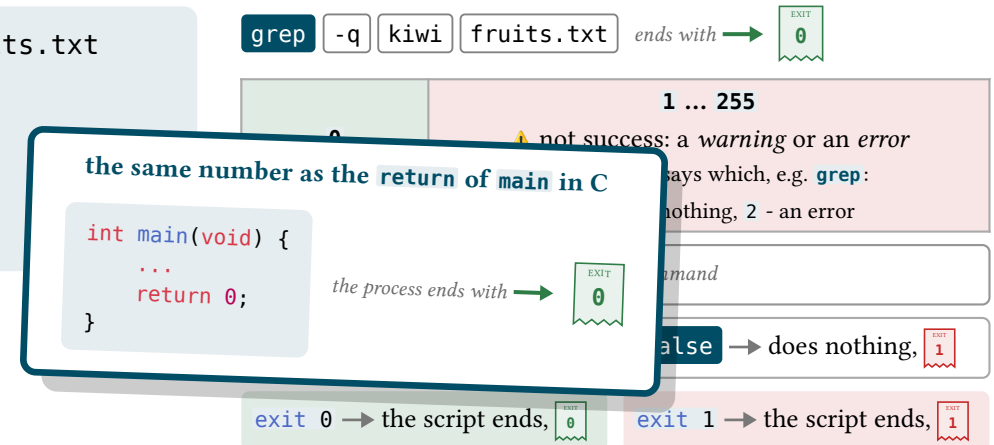
💡 `exit` without a number - the exit code of the last command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*

```
grep -q kiwi fruits.txt
echo $?
true
false
exit 0
exit 1
```



💡 **`exit`** without a number - the exit code of the last command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*, `$?` holds it

check.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt
3 echo "kiwi: $?"
4 grep -q mango fruits.txt
5 echo "mango: $?"
```

variables

```
$ ./check.sh
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `grep -q kiwi fruits.txt` → EXIT `0`

💡 `kiwi` is in the file: `grep` ends with `0`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*, `$?` holds it

check.sh

```
1  #!/bin/bash
2  grep -q kiwi fruits.txt
3  echo "kiwi: $?"
4  grep -q mango fruits.txt
5  echo "mango: $?"
```

variables

`$?` = `0`

```
$ ./check.sh
kiwi: 0
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `echo "kiwi: 0"` → EXIT `0`

💡 `$?` - the exit code of the command before, `grep`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*, `$?` holds it

check.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt
3 echo "kiwi: $?"
4 grep -q mango fruits.txt
5 echo "mango: $?"
```

variables

`$?` = `0`

```
$ ./check.sh
kiwi: 0
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `grep -q mango fruits.txt` → EXIT `1`

💡 `$?` is now the exit code of `echo`; no `mango`: `grep` ends with `1`, a warning, not an error

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Exit Codes

Every command ends with a number, `0` means *success*, `$?` holds it

check.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt
3 echo "kiwi: $?"
4 grep -q mango fruits.txt
5 echo "mango: $?"
```

variables

`$?` = `1`

```
$ ./check.sh
kiwi: 0
mango: 1
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `echo "mango: 1"` → EXIT `0`

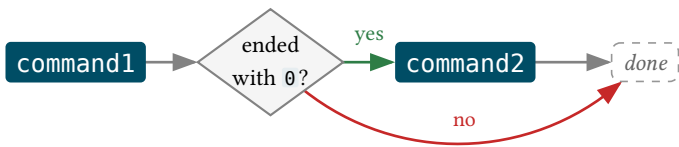
💡 `$?` - `1`, the exit code of **`grep`**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Chain Commands: `&&` and `||`

Run a command depending on the exit code of the one before

```
command1 && command2
command1 || command2
mkdir Movies && cd Movies || echo "failed"
```



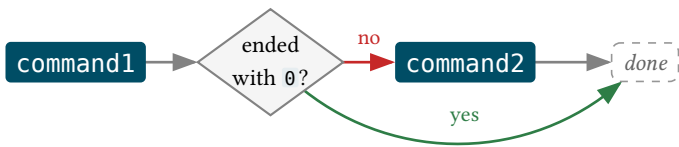
💡 `a && b` - `b` runs only if `a` ended with `0`, *and then*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Chain Commands: `&&` and `||`

Run a command depending on the exit code of the one before

```
command1 && command2
command1 || command2
mkdir Movies && cd Movies || echo "failed"
```



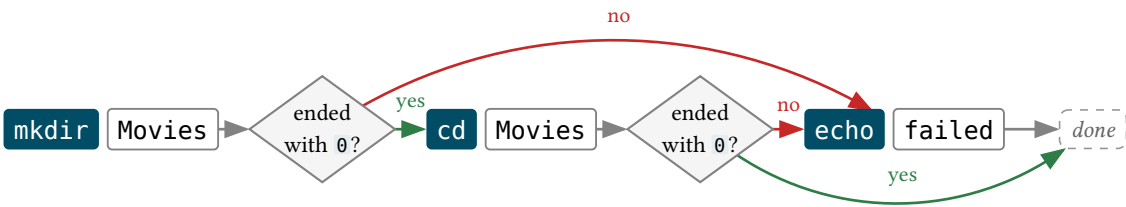
💡 `a || b` - `b` runs only if `a` did **not** end with `0`, *or else*

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Chain Commands: `&&` and `||`

Run a command depending on the exit code of the one before

```
command1 && command2
command1 || command2
mkdir Movies && cd Movies || echo "failed"
```



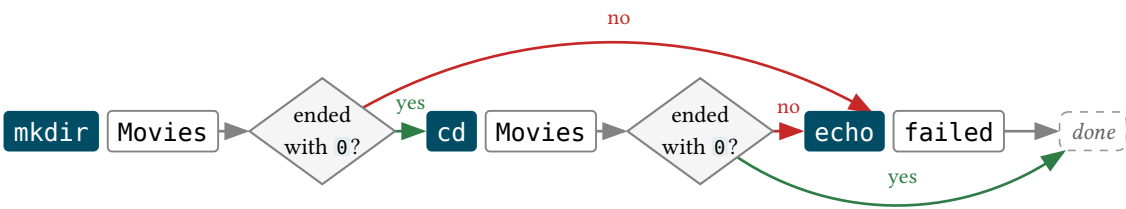
💡 they chain, from left to right: `echo` runs if `mkdir` **or** `cd` failed

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Chain Commands: `&&` and `||`

Run a command depending on the exit code of the one before

```
command1 && command2
command1 || command2
mkdir Movies && cd Movies || echo "failed"
```



💡 they chain, from left to right: `echo` runs if `mkdir` **or** `cd` failed

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

&& and ||

The exit code of the first command decides if the second one runs

chain.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt && echo "kiwi"
3 grep -q mango fruits.txt && echo "mango"
4 grep -q mango fruits.txt || echo "no mango"
```

```
$ ./chain.sh
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `grep -q kiwi fruits.txt` → EXIT 0

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

&& and ||

The exit code of the first command decides if the second one runs

chain.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt && echo "kiwi"
3 grep -q mango fruits.txt && echo "mango"
4 grep -q mango fruits.txt || echo "no mango"
```

```
$ ./chain.sh
kiwi
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `echo "kiwi"` → EXIT `0`

💡 `0` - `&&` runs the next command

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

&& and ||

The exit code of the first command decides if the second one runs

chain.sh

```
1  #!/bin/bash
2  grep -q kiwi fruits.txt && echo "kiwi"
3  grep -q mango fruits.txt && echo "mango"
4  grep -q mango fruits.txt || echo "no mango"
```

```
$ ./chain.sh
kiwi
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `grep -q mango fruits.txt` → EXIT 1

💡 not `0` - `&&` skips `echo "mango"`, the shell goes to the next line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

&& and ||

The exit code of the first command decides if the second one runs

chain.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt && echo "kiwi"
3 grep -q mango fruits.txt && echo "mango"
4 grep -q mango fruits.txt || echo "no mango"
```

```
$ ./chain.sh
kiwi
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

the shell runs `grep -q mango fruits.txt` → EXIT 1

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

&& and ||

The exit code of the first command decides if the second one runs

chain.sh

```
1 #!/bin/bash
2 grep -q kiwi fruits.txt && echo "kiwi"
3 grep -q mango fruits.txt && echo "mango"
4 grep -q mango fruits.txt || echo "no mango"
```

```
$ ./chain.sh
kiwi
no mango
```

fruits.txt

```
banana
apple
cherry
apple
kiwi
banana
apple
```

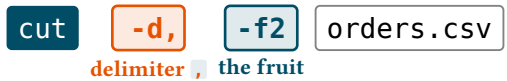
the shell runs `echo "no mango"` → EXIT 0

💡 not `0` - `||` runs it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command



orders.csv

```
name,fruit,count
alice,apple,3
bob,kiwi,1
carol,apple,2
alice,banana,5
dan,apple,1
alice,apple,2
```

stdout

```
fruit
apple
kiwi
apple
banana
apple
apple
```

💡 keep only the fruit column: we look for a *fruit*, not for a name

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command





stdin

```
fruit
apple
kiwi
apple
banana
apple
apple
```

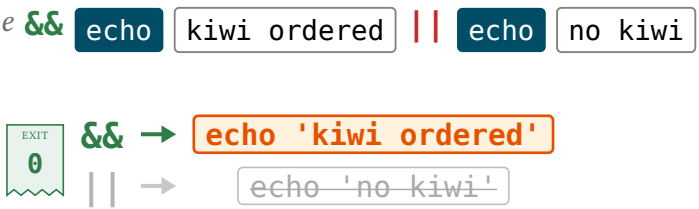
stdout

💡 `-q` (*quiet*) prints nothing, it only ends with `0` - found; the pipeline ends with the exit code of `grep`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command




```
$ cut -d, -f2 orders.csv | grep -q kiwi \
  && echo 'kiwi ordered' || echo 'no kiwi'
kiwi ordered
```

✅ `kiwi` is in the column: the exit code is `0`, `&&` runs

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command





stdin

```
fruit
apple
kiwi
apple
banana
apple
apple
```

stdout

💡 no `mango` in the column: `grep` ends with `1`, so does the pipeline

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command




```
$ cut -d, -f2 orders.csv | grep -q mango \
  && echo 'mango ordered' || echo 'no mango'
no mango
```

💡 the exit code is `1`: `&&` skips its command, `||` runs

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command



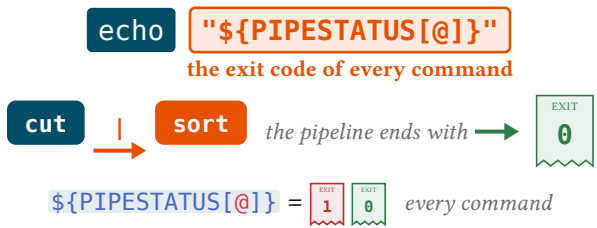

```
$ cut -d, -f2 nope.csv | sort
cut: nope.csv: No such file or directory
$ echo $?
0
```

⚠ `cut` failed, but `sort` worked: the pipeline ends with `0`, the error is **hidden**

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Exit Code of a Pipeline

The exit code of the **last** command




```
$ cut -d, -f2 nope.csv | sort
cut: nope.csv: No such file or directory
$ echo "${PIPESTATUS[@]}"
1 0
```

💡 `PIPESTATUS` (`bash`) keeps them all; `set -o pipefail` - the pipeline ends with the last **failed** one

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Text Editors

Writing code is editing text files

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 2 - *Basic Commands and Directory Hierarchy*
     - Section 2.12 - *Text Editors*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.6.1 - *Scrierea codului sursă*
     - Section 6.6.2 - *Editoare în mod text*
3. the built-in tutorials
   - `vimtutor` - 30 minutes of `vim`
   - `hx --tutor` - the `helix` tutorial
   - `Ctrl` + `h` then `t` inside `emacs`

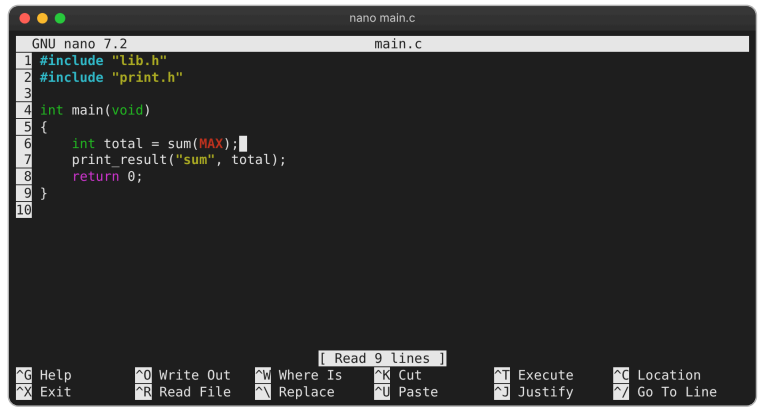
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Terminal Editors

Edit a file inside the terminal: on a server, over `ssh`, with no windows

`nano`

very basic, the keys are listed at the bottom



💡 `nano` (a free clone of `pico`): what you see is what you get, the default editor on many distributions

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Terminal Editors

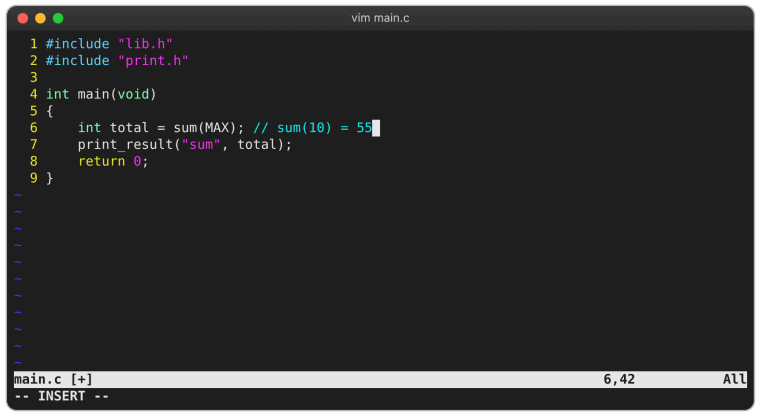
Edit a file inside the terminal: on a server, over `ssh`, with no windows

`nano`

very basic, the keys are listed at the bottom

`vi` / `vim` / `nvim`

modal, available everywhere



💡 `vi` (1976), `vim` (*Vi IMproved*, 1991), `neovim` (2014): most IDEs have a `vim` mode, and `j` `k` also scroll in `less` and `man`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Terminal Editors

Edit a file inside the terminal: on a server, over `ssh`, with no windows

`nano`

very basic, the keys are listed at the bottom

`vi` / **`vim`** / **`nvim`**

modal, available everywhere

`emacs`

key chords, extensible in Lisp



💡 **`emacs`** (1976): not modal, every command is a `Ctrl` or `Alt` chord

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Terminal Editors

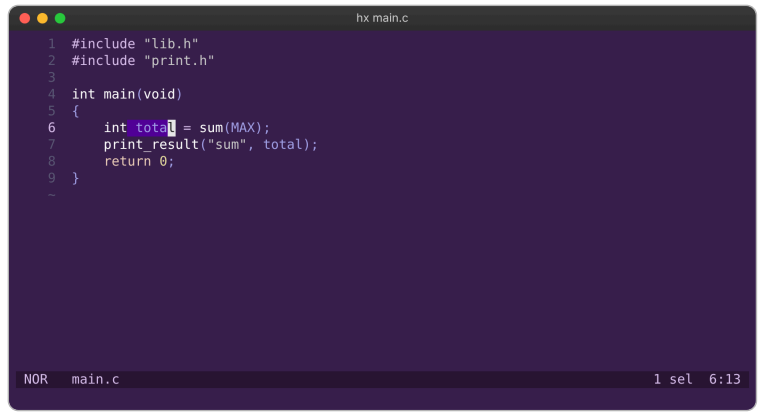
Edit a file inside the terminal: on a server, over `ssh`, with no windows

> **`nano`**
> *very basic*, the keys are listed at the bottom

> **`vi` / `vim` / `nvim`**
> *modal*, available everywhere

> **`emacs`**
> key chords, extensible in Lisp

> **`helix` (`hx`)**
> *modal*, select first, then act

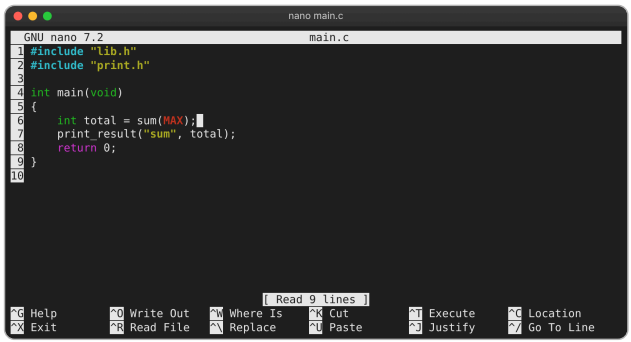


💡 `helix` (2021, written in Rust): `vim`-like, with a language server client built in

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

nano

The keys are always on the screen: `^` means Ctrl, `M-` means Alt



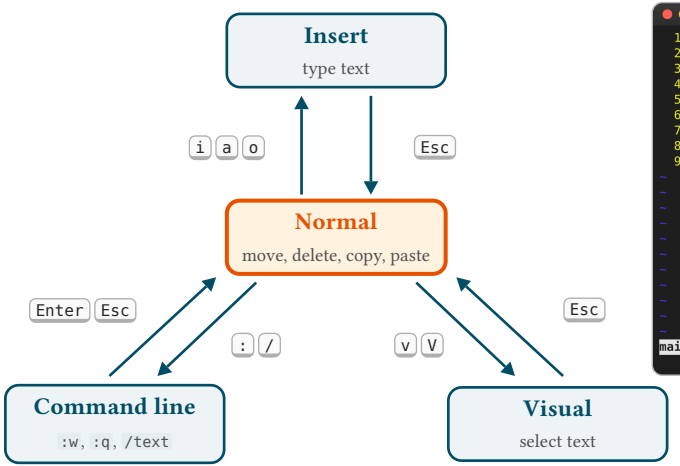

```
$ nano main.c
```

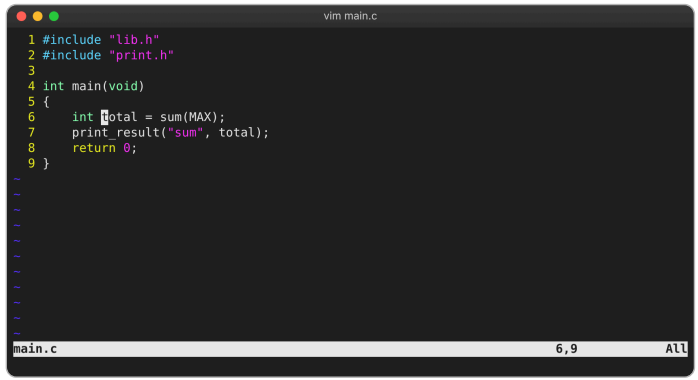
key	action
move	
← ↓ ↑ →	the arrow keys
Ctrl + A / Ctrl + E	start / end of the line
files	
Ctrl + O	save (*write out*)
Ctrl + X	exit
edit	
Ctrl + K	cut the line
Ctrl + U	paste
Alt + U	undo
search	
Ctrl + W	search (*where is*)
Ctrl + \	search and replace
Ctrl + _	go to a line (_)

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`vim` is Modal

The same key does different things in different *modes*







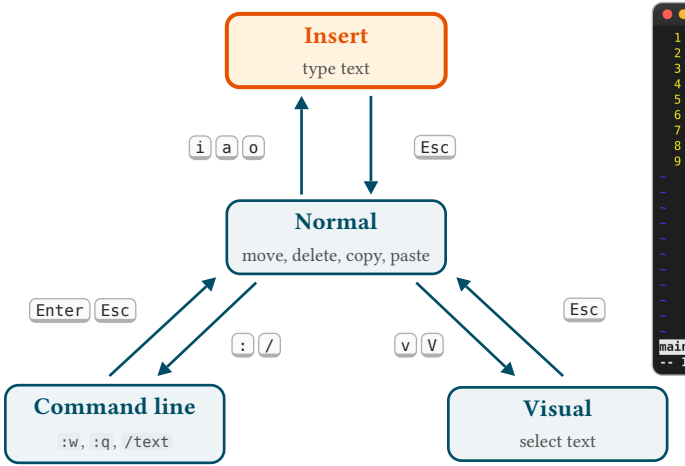
Normal: the last line is empty, the cursor is on a character

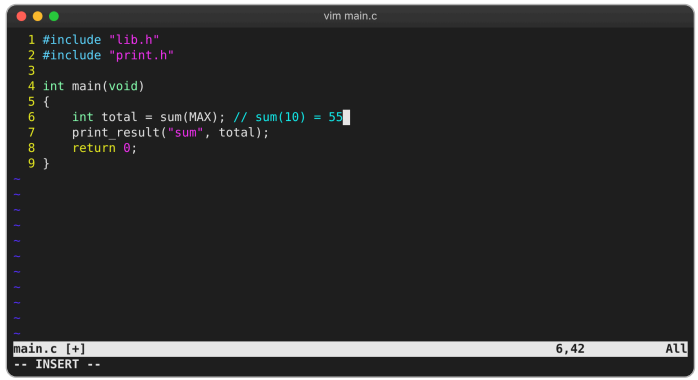
💡 `vim` starts in **Normal** mode: the keys are **commands**, `d` does not type a `d`, it deletes

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`vim` is Modal

The same key does different things in different *modes*







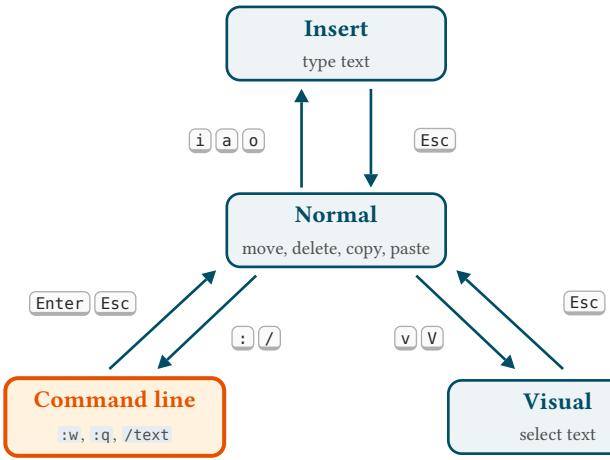
Insert: `-- INSERT --` on the last line, the typed text goes into the file

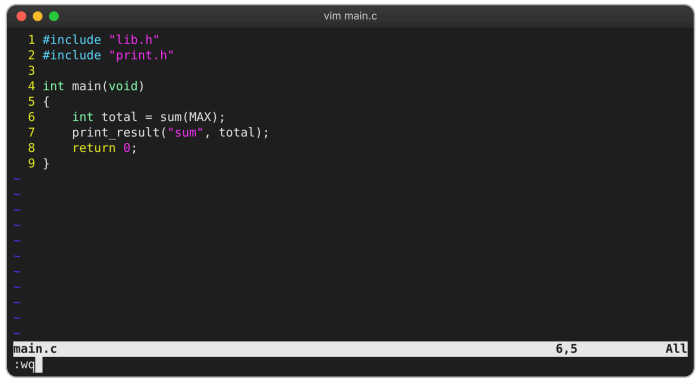
💡 `i` inserts before the cursor, `a` appends after it, `o` opens a new line; `Esc` goes back

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`vim` is Modal

The same key does different things in different *modes*







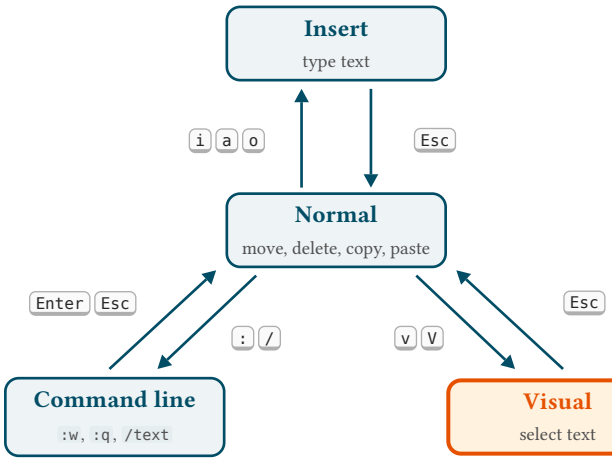
Command line: the cursor moves to the last line, `Enter` runs `:wq`

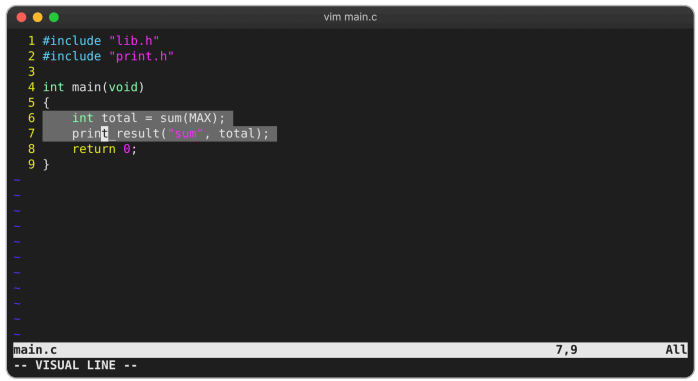
💡 `:` starts a command, e.g. `:w` save, `:q` quit, `:wq` both, `:q!` quit without saving

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

vim is Modal

The same key does different things in different *modes*







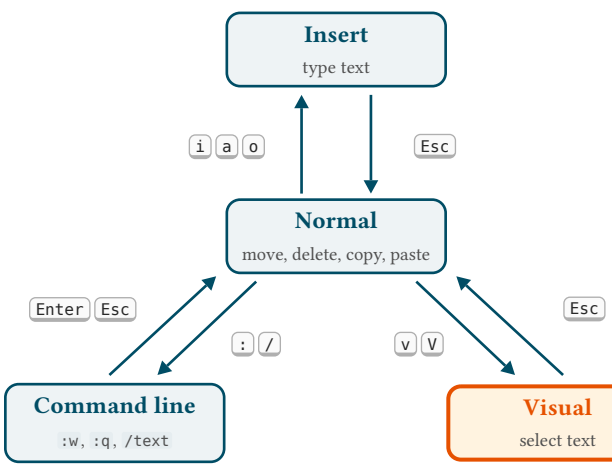
Visual: `-- VISUAL LINE --`, lines 6 and 7 are selected

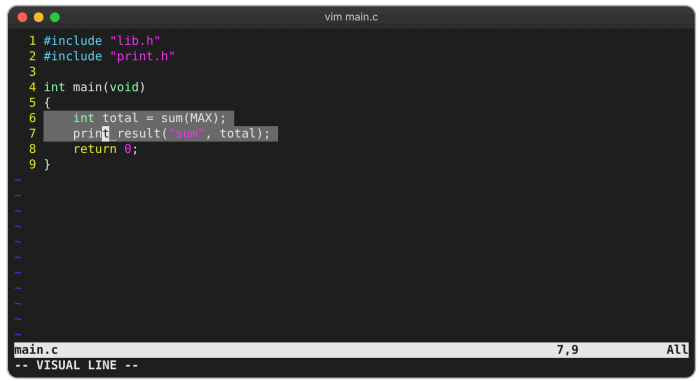
💡 `v` selects characters, `V` whole lines; then `d` deletes or `y` copies the selection

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

vim is Modal

The same key does different things in different *modes*







Visual: `-- VISUAL LINE --`, lines 6 and 7 are selected

💡 always look at the **last line** to know the mode; lost? press Esc a few times: Normal mode

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

start

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

Normal mode, the cursor is on `int`, line 6

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: `l` or `→`

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`l` or `→` one character to the right

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: `j` or `↓`

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`j` or `↓` one line down, the same column

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed:  or

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`k` or `↑` one line up

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: h or ←

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`h` or `←` one character to the left

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: **`w`**

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`w` → to the next **word**: `total`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: **$**

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`$` → to the end of the line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters

`h` `j` `k` `l`

or the arrow keys

← ↓ ↑ →

- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: **`0`**

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`0` ← to the start of the line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: **G**

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`G` ↓ to the last line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Moving in `vim`

In Normal mode, the hands stay on the letters



- `h` is the leftmost, `l` the rightmost
- a count repeats a move: `5` `j` is 5 lines down
- `h` `j` `k` `l` also work in **`helix`**, and in VS Code and Zed with their **`vim`** modes; in **`less`** and **`man`**, only `j` `k` (scroll)

pressed: **g** **g**

```
1  #include "lib.h"
2  #include "print.h"
3
4  int main(void)
5  {
6      int total = sum(MAX);
7      print_result("sum", total);
8      return 0;
9  }
```

`g` `g` ↑ to the first line

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

`vim` Survival Keys

In Normal mode

save and quit

`:w`	save
`:q`	quit
`:wq`	save and quit
`:q!`	quit, don't save

move

`h` `j` `k` `l`	← ↓ ↑ →
`←` `↓` `↑` `→`	also work
`w` / `b`	next / prev word
`0` / `$`	start / end of line
`g` `g` / `G`	first / last line

edit

`x`	delete a char
`d` `d`	delete the line
`u`	undo
`Ctrl` + `R`	redo

insert text

`i` / `a`	before / after
`o` / `O`	new line ↓ / ↑
`Esc`	back to Normal

search

`/text`	search forward
`n` / `N`	next / prev one
`4` `2` `G`	go to line 42

copy and paste

`v`	select (*Visual*)
`y` / `d`	copy / cut
`p`	paste

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
d (verb) w (motion)	**d**elete a **w**ord	`int total = sum(MAX);`	`int = sum(MAX);`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
d (verb) w (motion)	**d**elete a **w**ord	`int total = sum(MAX);`	`int = sum(MAX);`
d (verb) 3 (count) w (motion)	**d**elete **3** **w**ords	`int total = sum(MAX);`	`int (MAX);`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
d (verb) w (motion)	**d**elete a **w**ord	`int total = sum(MAX);`	`int = sum(MAX);`
d (verb) 3 (count) w (motion)	**d**elete **3** **w**ords	`int total = sum(MAX);`	`int (MAX);`
c (verb) $ (motion)	**c**hange to **line end**	`int total = sum(MAX);`	`int total =` Insert

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
d (verb) w (motion)	**d**elete a **w**ord	`int total = sum(MAX);`	`int = sum(MAX);`
d (verb) 3 (count) w (motion)	**d**elete **3** **w**ords	`int total = sum(MAX);`	`int (MAX);`
c (verb) $ (motion)	**c**hange to **line end**	`int total = sum(MAX);`	`int total =` Insert
c (verb) i " (text object)	**c**hange **inside** "	`print_result("sum", total);`	`print_result("", total);` Insert

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
d (verb) w (motion)	**d**elete a **w**ord	`int total = sum(MAX);`	`int = sum(MAX);`
d (verb) 3 (count) w (motion)	**d**elete **3** **w**ords	`int total = sum(MAX);`	`int (MAX);`
c (verb) $ (motion)	**c**hange to **line end**	`int total = sum(MAX);`	`int total =` Insert
c (verb) i " (text object)	**c**hange **inside "**	`print_result("sum", total);`	`print_result("", total);` Insert
y (verb) i ((text object)	**y**ank (copy) **inside ()**	`int total = sum(MAX);`	`MAX` is copied, p pastes it

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `vim`

A command is a sentence:

verb + **(count)** + **motion** or **text object**

keys	meaning	before	after
`d` `w` — verb motion	delete a word	`int total = sum(MAX);`	`int = sum(MAX);`
`d` `3` `w` — verb count motion	delete 3 words	`int total = sum(MAX);`	`int (MAX);`
`c` `$` — verb motion	change to line end	`int total = sum(MAX);`	`int total =` Insert
`c` `i` `"` — verb text object	change inside "	`print_result("sum", total);`	`print_result("", total);` Insert
`y` `i` `(` — verb text object	yank (copy) inside ()	`int total = sum(MAX);`	`MAX` is copied, `p` pastes it

verbs: `d` delete, `c` change, `y` yank, `>` indent | doubled, it acts on the line: `d d`, `y y`, `> >`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

emacs

Not modal: commands are key chords, `C-` is `Ctrl`, `M-` is `Alt` (*Meta*)



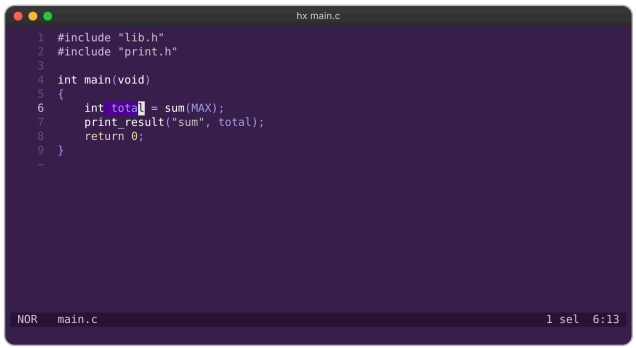

```
$ emacs -nw main.c
```

key	action
move	
Ctrl + B / Ctrl + F	← → *back* / *forward*
Ctrl + P / Ctrl + N	↑ ↓ *previous* / *next* line
Ctrl + A / Ctrl + E	start / end of the line
files	
Ctrl + X Ctrl + F	open a file
Ctrl + X Ctrl + S	save
Ctrl + X Ctrl + C	quit
edit	
Ctrl + K	cut (*kill*) to the end of the line
Ctrl + Y	paste (*yank*)
Ctrl + /	undo
other	
Ctrl + S	search
Ctrl + G	cancel the current command
Alt + X	run any command by name

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

helix

Like `vim`, but **select first**, then act



`w` selected `total` and the space after it: you **see** what will change

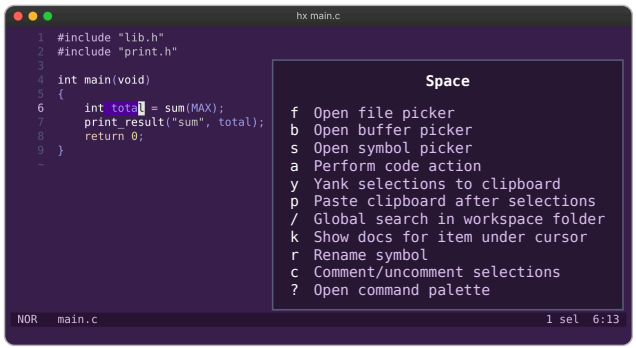
	vim	helix
move ← ↓ ↑ →	`h` `j` `k` `l`	`h` `j` `k` `l`
delete a word	`d` `w`	`w` `d`
delete the line	`d` `d`	`x` `d`
change inside `"`	`c` `i` `"`	`m` `i` `"` `c`
select the next match		`*` `n`
one cursor per match		`%` `s`
add a cursor below		`C`
go to definition	*LSP setup*	`g` `d`
show documentation	*LSP setup*	`Space` `k`
save / quit	`:w` `:q`	`:w` `:q`

`Space` opens a menu with all the commands

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

helix

Like `vim`, but **select first**, then act



Space opens a menu: the next key picks the command

	vim	helix
move ← ↓ ↑ →	h j k l	h j k l
delete a word	d w	w d
delete the line	d d	x d
change inside "	c i "	m i " c
select the next match		* n
one cursor per match		% s
add a cursor below		C
go to definition	*LSP setup*	g d
show documentation	*LSP setup*	Space k
save / quit	:w :q	:w :q

Space opens a menu with all the commands

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
w (motion) d (verb)	select a **w**ord, **d**elete it	`int total= sum(MAX);`	`int = sum(MAX);`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
`w` (motion) `d` (verb)	select a **w**ord, **d**elete it	`int total = sum(MAX);`	`int  = sum(MAX);`
`t` `;` (motion) `d` (verb)	select **till** `;`, **d**elete it	`int total = sum(MAX);`	`int ;`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
w (motion) d (verb)	select a **w**ord, **d**elete it	`int total = sum(MAX);`	`int = sum(MAX);`
t ; (motion) d (verb)	select **till** `;`, **d**elete it	`int total = sum(MAX);`	`int ;`
v (extend) w w (motion) d (verb)	**visual 2 words**, **d**elete	`int total = sum(MAX);`	`int sum(MAX);`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
w (motion) d (verb)	select a **w**ord, **d**elete it	`int total= sum(MAX);`	`int = sum(MAX);`
t ; (motion) d (verb)	select **till** `;`, **d**elete it	`int total = sum(MAX);`	`int ;`
v (extend) w w (motion) d (verb)	**visual 2 words**, **d**elete	`int total = sum(MAX);`	`int sum(MAX);`
m i " (text object) c (verb)	**match inside** `"`, **c**hange	`print_result("sum", total);`	`print_result("", total);` Insert

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
`w` motion, `d` verb	select a **word**, **delete** it	`int total= sum(MAX);`	`int = sum(MAX);`
`t` `;` motion, `d` verb	select **till** `;`, **delete** it	`int total = sum(MAX);`	`int ;`
`v` extend, `w` `w` motion, `d` verb	**visual 2 words**, **delete**	`int total = sum(MAX);`	`int sum(MAX);`
`m` `i` `"` text object, `c` verb	**match inside** `"`, change	`print_result("sum", total);`	`print_result("", total);` Insert
`%` file, `s` `total` ⏎ each match, `c` verb	**every** `total` in the file, change all	`int total = sum(MAX);` `print_result("sum", total);`	`int count = sum(MAX);` `print_result("sum", count);`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

The Language of `helix`

A command is: first **select**, then **act** on the selection

motion or **text object** or **selection** → **verb**

keys	meaning	selected	after
`w` motion, `d` verb	select a **w**ord, **d**elete it	`int total= sum(MAX);`	`int = sum(MAX);`
`t` `;` motion, `d` verb	select **till** `;`, **d**elete it	`int total = sum(MAX);`	`int ;`
`v` extend, `w` `w` motion, `d` verb	**visual 2 words**, **d**elete	`int total = sum(MAX);`	`int sum(MAX);`
`m` `i` `"` text object, `c` verb	**match inside** `"`, **c**hange	`print_result("sum", total);`	`print_result("", total);` Insert
`%` file, `s` `total` `⏎` each match, `c` verb	**every** `total` in the file, **c**hange all	`int total = sum(MAX);` `print_result("sum", total);`	`int count = sum(MAX);` `print_result("sum", count);`

`vim`: **verb** then **motion** (`d` `w`) | `helix`: **selection** then **verb** (`w` `d`): you see what will change

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Graphical Editors

Code editors with windows, tabs and a mouse

Bibliography

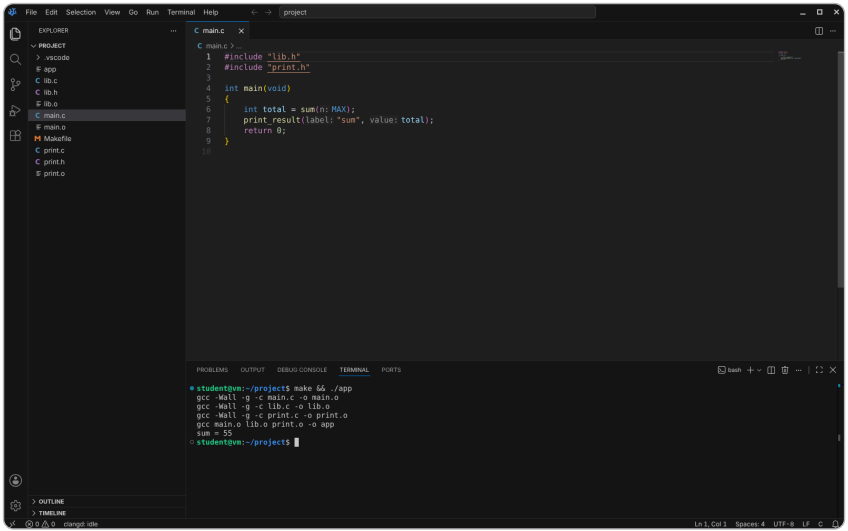
For this section

1. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.6.3 - *Editoare în interfață grafică*
2. *Visual Studio Code documentation*, online
3. *VSCodium*, online
4. *Zed documentation*, online

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

VS Code / VSCodium

The most used code editor, and its open source build



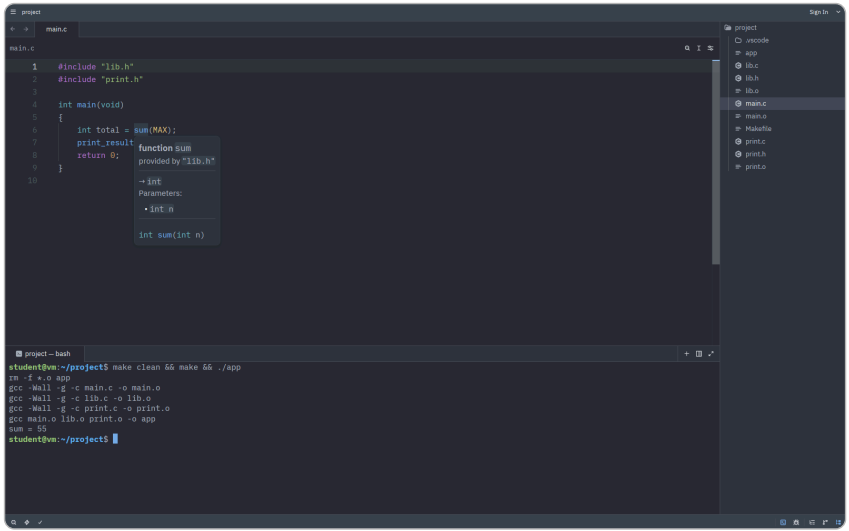
- by Microsoft, free of charge, **not** open source
- a **browser**: built with Electron (Chromium + Node.js)
- an **extension** for everything
- VSCodium: the open source code, without telemetry
  - its extensions come from *Open VSX*
  - ⚠️ some Microsoft extensions do not work: C/C++ (the debugger), Remote - SSH

`vim` keys: the Vim or VSCode Neovim extension

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Zed

A native editor, drawn by the GPU



- written in Rust
- **not** a browser: a native app, drawn by the **GPU**, very fast
- language servers, terminal, debugger built in

vim and **helix** keys built in: `vim_mode`, `helix_mode`

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

We talked about

- Pipelines
- Line Filters
  - `cat`, `tac`, `head`, `tail`, `sort`, `uniq`, `nl`
- Search
  - regular expressions, `grep`
- Change Text
  - `cut`, `tr`, `sed`, `rev`
- Compute
  - `wc`, `uniq -c`, `awk`
- A Pipeline, Step by Step
- Exit Codes
- Text Editors
  - `nano`, `vim`, `emacs`, `helix`
- Graphical Editors
  - VS Code / VSCodium, Zed

 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0