Introduction to Operating Systems

7. Software Development

2026-11-13

USO Team

Bibliography

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.1 - *The C Compiler*
     - Section 15.2 - *make*
     - Section 15.4 - *Scripting Languages*
     - Section 15.5 - *Java*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
3. **Richard M. Stallman, Roland McGrath, Paul D. Smith**, *GNU Make Manual*, online
   - Chapter 2 - *An Introduction to Makefiles*
4. *Language Server Protocol Specification*, online
5. *Debug Adapter Protocol Specification*, online
6. **Scott Chacon, Ben Straub**, *Pro Git*, 2nd Edition, Apress, 2014, online
   - Chapters 1 - 3

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

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan and P. J. Plauger, *The Elements of Programming Style*

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

Software Development

- Compiled & Interpreted Languages
  - compiler, interpreter, bytecode
- Integrated Development Environments
- Language Servers
- Debuggers
- Debug Adapters
- The C Compilation Pipeline
  - preprocessor, compiler, assembler, linker
- Building with `make`
  - rules, step by step
- A `Makefile` for a Project
  - automatic variables, pattern rules
- Special Targets
  - `.PHONY`, `clean`, `all`, `run`
- Header Dependencies
- Version Control with Git
  - commits, commit vs push, branches
- From an Editor to an IDE
  - LSP + DAP

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

Compiled & Interpreted Languages

Who turns the source code into machine code, and when?

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.4 - *Scripting Languages*
     - Section 15.5 - *Java*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.2 - *Limbaje de programare*
     - Sections 6.3 - 6.5 - *Limbaje compilate*, *Limbaje interpretate*, *Limbaje hibride*

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

Machine Code

We write the source code as text, the processor runs only bytes

source code (text), what we write

```c
for (int i = 0; i < 3; i++) {
    printf("Hello, %d!\n", i);
}
```



machine code (bytes), what the CPU runs

```
55 48 89 e5 48 83 ec 10
c7 45 fc 00 00 00 00 eb
1d 8b 45 fc 89 c6 48 8d
05 00 00 00 00 48 89 c7
…
```

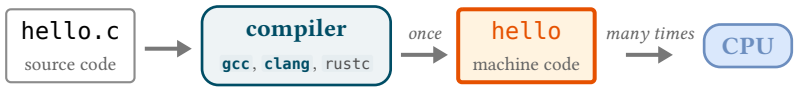
💡 every processor has its own **instruction set**: x86-64, ARM64, RISC-V, …

💡 someone has to **translate** the source code to machine code: **who** does it and **when**?

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

Compiled Languages

Translate everything once, run the result many times

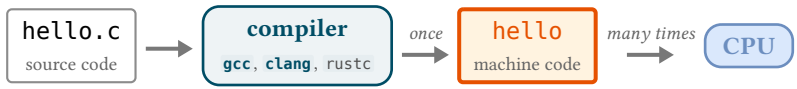



```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
$ file hello
hello: ELF 64-bit LSB pie
executable, ...
```

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

Compiled Languages

Translate everything once, run the result many times



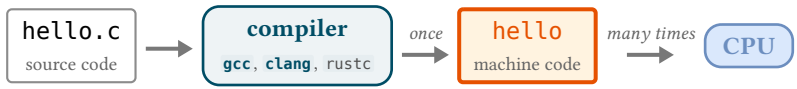

```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
$ file hello
hello: ELF 64-bit LSB pie
executable, ...
```

✅ **fast**: the CPU runs the program directly
✅ many errors (syntax, types) are found **before** running
✅ ship only the executable, not the source

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

Compiled Languages

Translate everything once, run the result many times




```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
$ file hello
hello: ELF 64-bit LSB pie
executable, ...
```

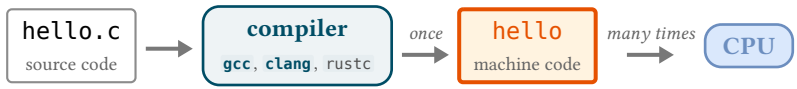
✅ **fast**: the CPU runs the program directly
✅ many errors (syntax, types) are found **before** running
✅ ship only the executable, not the source

❌ one executable per **processor** and **OS**
❌ compile again after **every** change

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

Compiled Languages

Translate everything once, run the result many times




```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
$ file hello
hello: ELF 64-bit LSB pie
executable, ...
```

✅ **fast**: the CPU runs the program directly
✅ many errors (syntax, types) are found **before** running
✅ ship only the executable, not the source

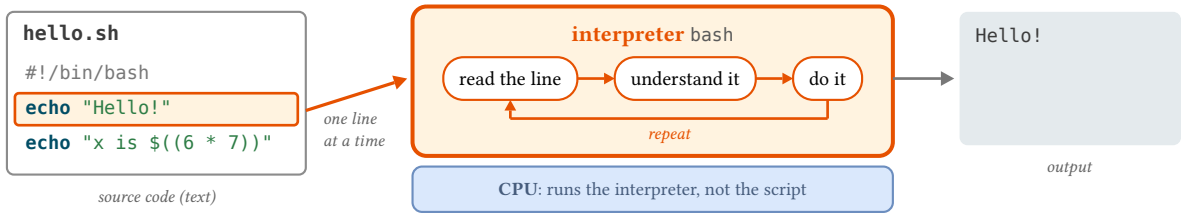
❌ one executable per **processor** and **OS**
❌ compile again after **every** change

C **C++** **Rust** **Go** **Zig** **Fortran**

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

Interpreted Languages

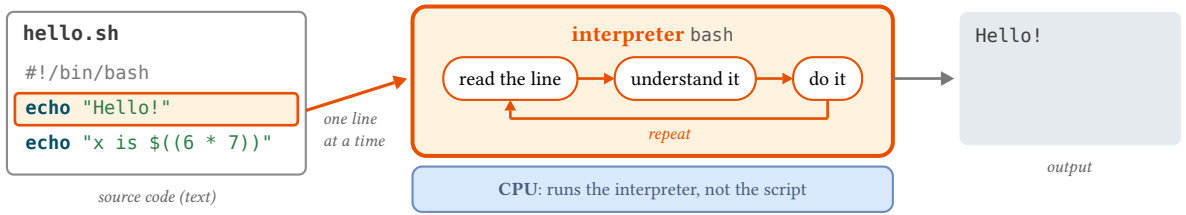
A program (the interpreter) reads the source and does what it says



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

Interpreted Languages

A program (the interpreter) reads the source and does what it says

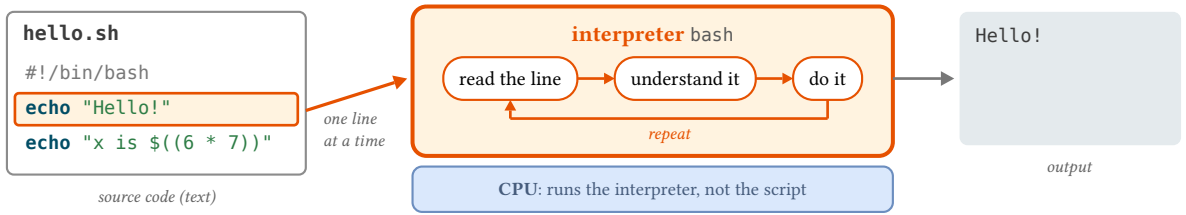



```
$ ./hello.sh
Hello!
x is 42
$ file hello.sh
hello.sh: Bourne-Again shell script, ...
$ file /bin/bash
/bin/bash: ELF 64-bit LSB pie executable, ...
```

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

Interpreted Languages

A program (the interpreter) reads the source and does what it says



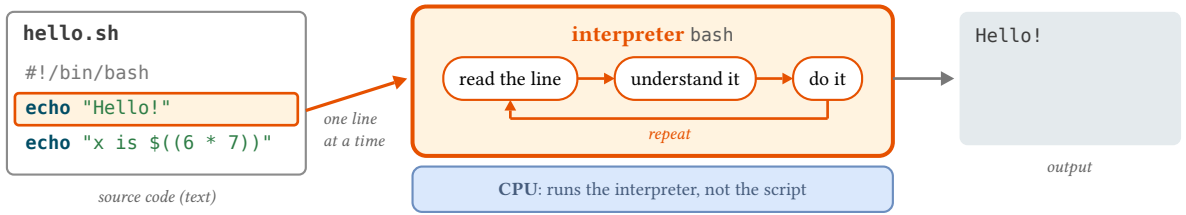

```
$ ./hello.sh
Hello!
x is 42
$ file hello.sh
hello.sh: Bourne-Again shell script, ...
$ file /bin/bash
/bin/bash: ELF 64-bit LSB pie executable, ...
```

✅ **portable**: runs wherever the interpreter runs
✅ edit and run, no build step

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

Interpreted Languages

A program (the interpreter) reads the source and does what it says




```
$ ./hello.sh
Hello!
x is 42
$ file hello.sh
hello.sh: Bourne-Again shell script, ...
$ file /bin/bash
/bin/bash: ELF 64-bit LSB pie executable, ...
```

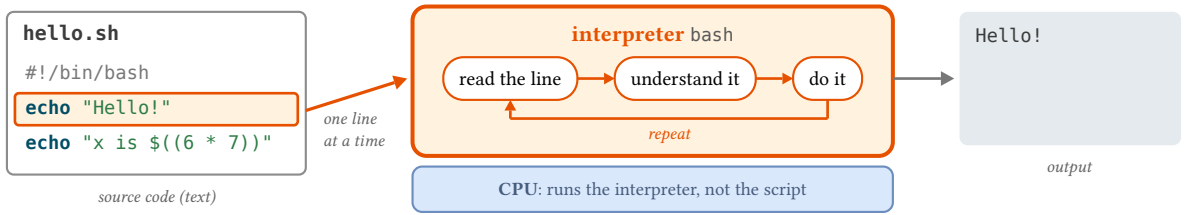
✅ **portable**: runs wherever the interpreter runs
✅ edit and run, no build step

❌ **slower**: the source is translated at every run
❌ errors are found only when that line **runs**
❌ ship the source code

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

Interpreted Languages

A program (the interpreter) reads the source and does what it says




```
$ ./hello.sh
Hello!
x is 42
$ file hello.sh
hello.sh: Bourne-Again shell script, ...
$ file /bin/bash
/bin/bash: ELF 64-bit LSB pie executable, ...
```

✅ **portable**: runs wherever the interpreter runs
✅ edit and run, no build step

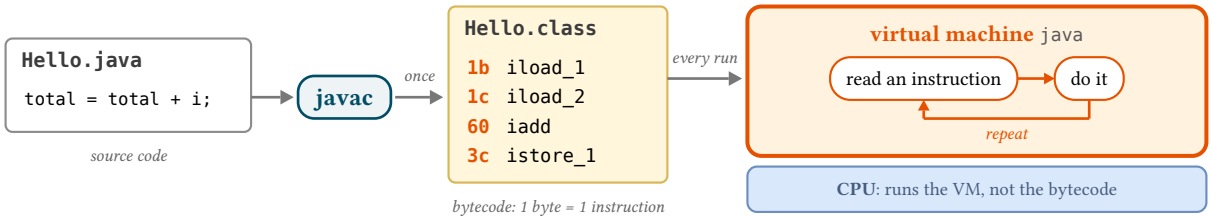
❌ **slower**: the source is translated at every run
❌ errors are found only when that line **runs**
❌ ship the source code

bash **sh** **zsh** **classic BASIC**

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

Bytecode and Virtual Machines

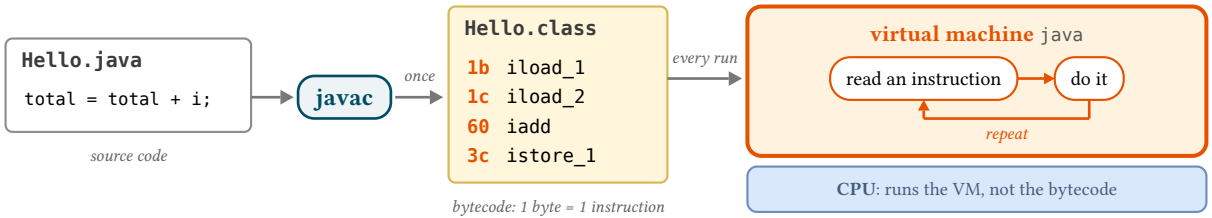
Hybrid languages: compile to an **intermediate** code, then interpret it



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

Bytecode and Virtual Machines

Hybrid languages: compile to an **intermediate** code, then interpret it



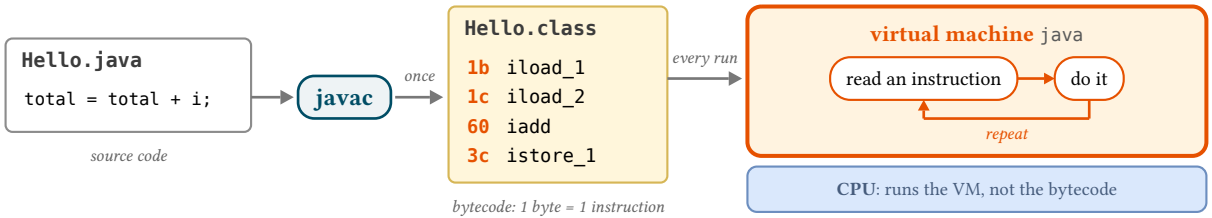

```
$ javac Hello.java
$ file Hello.class
Hello.class: compiled Java class data,
version 65.0
$ java Hello
Hello, 0!
...
```

- bytecode: machine code for an *imaginary* CPU, the VM
- the same `.class` file runs on any OS and CPU with a JVM

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

Bytecode and Virtual Machines

Hybrid languages: compile to an **intermediate** code, then interpret it



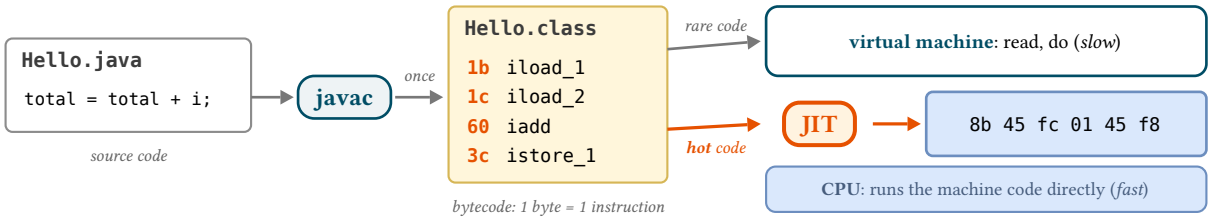

```
$ javap -c Hello
...
  10: iload_1
  11: iload_2
  12: iadd
  13: istore_1
...
```

- bytecode: machine code for an *imaginary* CPU, the VM
- the same `.class` file runs on any OS and CPU with a JVM

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

Bytecode and Virtual Machines

Hybrid languages: compile to an **intermediate** code, then interpret it



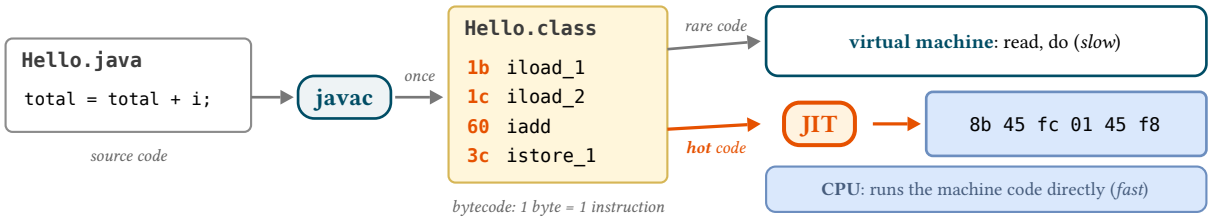

```
$ javap -c Hello
...
  10: iload_1
  11: iload_2
  12: iadd
  13: istore_1
...
```

- bytecode: machine code for an *imaginary* CPU, the VM
- the same `.class` file runs on any OS and CPU with a JVM
- **JIT** (*just in time*): the VM compiles the code that runs often to real machine code

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

Bytecode and Virtual Machines

Hybrid languages: compile to an **intermediate** code, then interpret it




```
$ javap -c Hello
...
  10: iload_1
  11: iload_2
  12: iadd
  13: istore_1
...
```

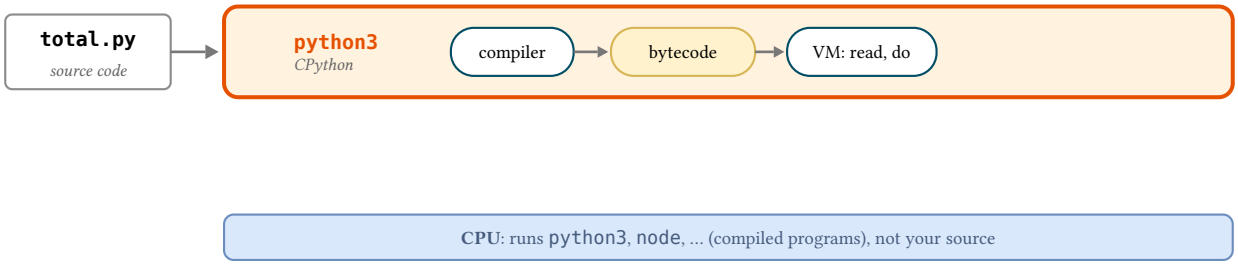
- bytecode: machine code for an *imaginary* CPU, the VM
- the same `.class` file runs on any OS and CPU with a JVM
- **JIT** (*just in time*): the VM compiles the code that runs often to real machine code

Java **Kotlin** **C#** **Python** **JavaScript**

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

Python is Compiled Too

... to bytecode, automatically, every time you start it



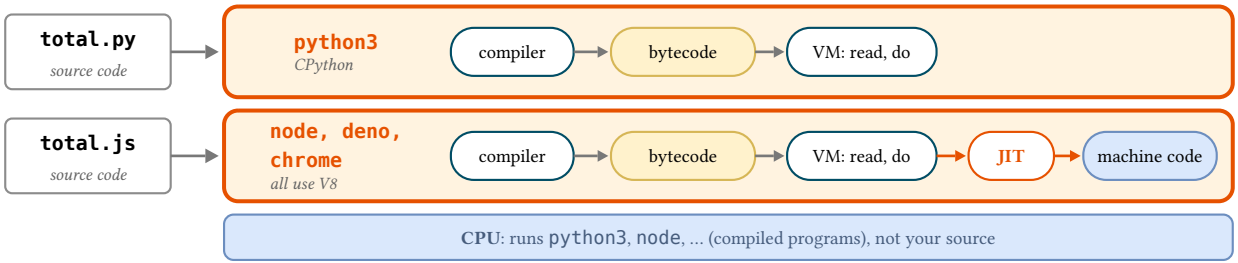

```
$ cat total.py
total = total + i
$ python3 -m dis total.py
  LOAD_NAME     0 (total)
  LOAD_NAME     1 (i)
  BINARY_OP     0 (+)
  STORE_NAME    0 (total)
```

- `python3` compiles the source to **bytecode** (cached in `__pycache__`), then its VM runs it
-

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

Python is Compiled Too

... to bytecode, automatically, every time you start it



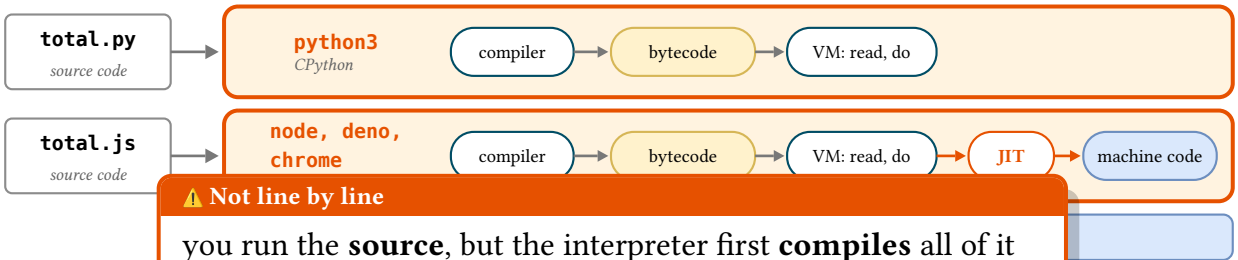

```
$ cat total.js
function add(total, i) { return total + i }
add(10, 3)
$ node --print-bytecode \
  --print-bytecode-filter=add total.js
  0b 04      Ldar a1
  3b 03 00   Add a0, [0]
  ae         Return
```

- `python3` compiles the source to **bytecode** (cached in `__pycache__`), then its VM runs it
- `node`, `deno` and `chrome` share the **V8** engine, that also JIT-compiles the hot code

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

Python is Compiled Too

... to bytecode, automatically, every time you start it



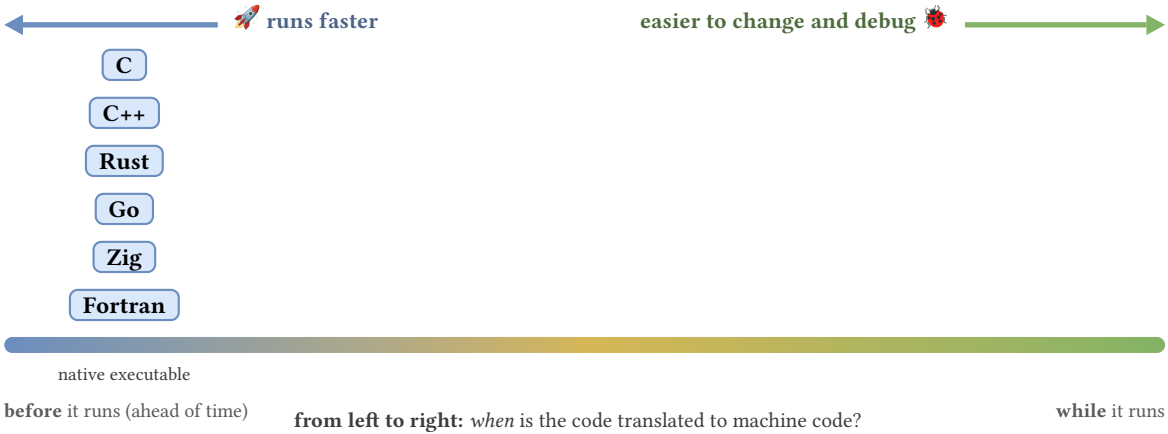

```
$ cat total.js
function add(total, i) { return total + i }
add(10, 3)
$ node --print-bytecode \
  --print-bytecode-filter=add total.js
  0b 04      Ldar a1
  3b 03 00   Add a0, [0]
  ae         Return
```

- **de** (cached in `__pycache__`), then its VM runs it
- `node`, `deno` and `chrome` share the **V8** engine, that also JIT-compiles the hot code

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

The Big Picture

The line between *compiled* and *interpreted* is blurry

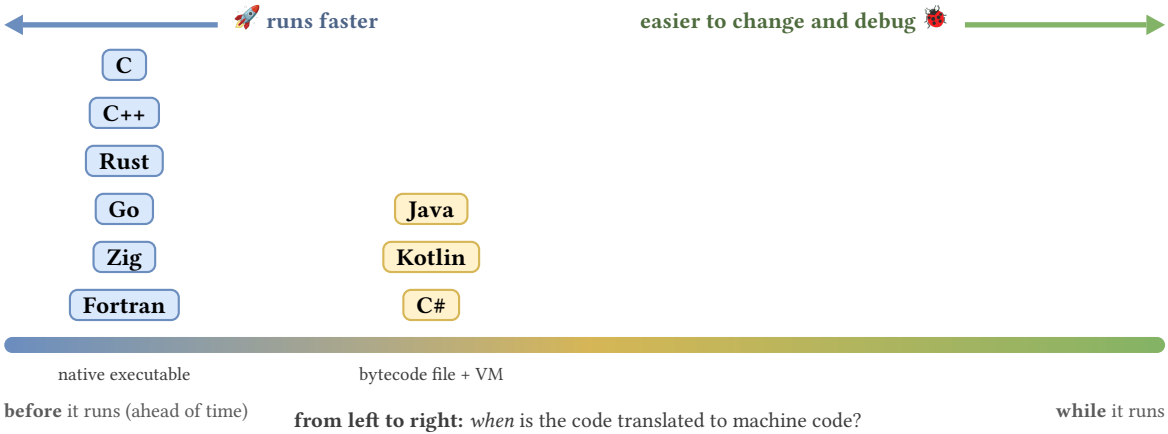


💡 compiled to a **native executable**: the CPU runs it directly

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

The Big Picture

The line between *compiled* and *interpreted* is blurry

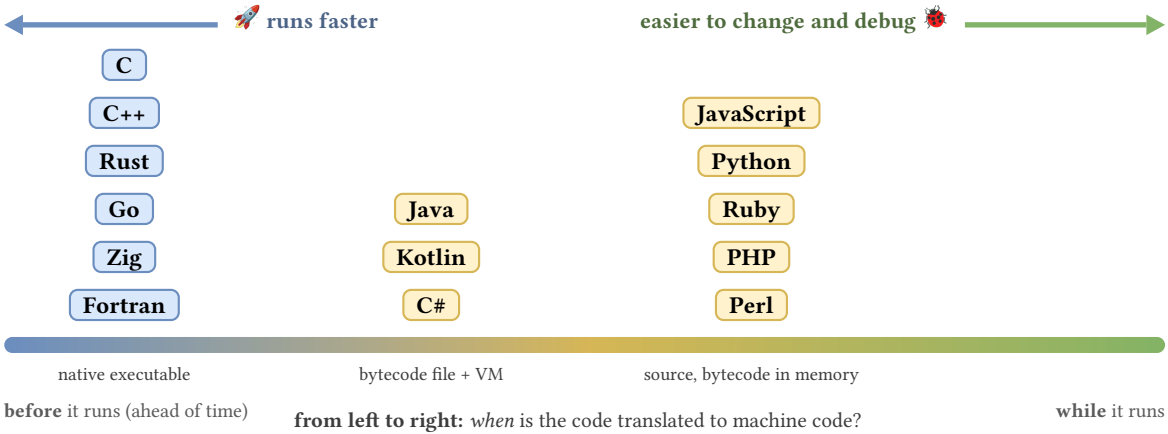


💡 compiled to a **bytecode file**: a virtual machine (JVM, .NET) runs it, with a JIT

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

The Big Picture

The line between *compiled* and *interpreted* is blurry

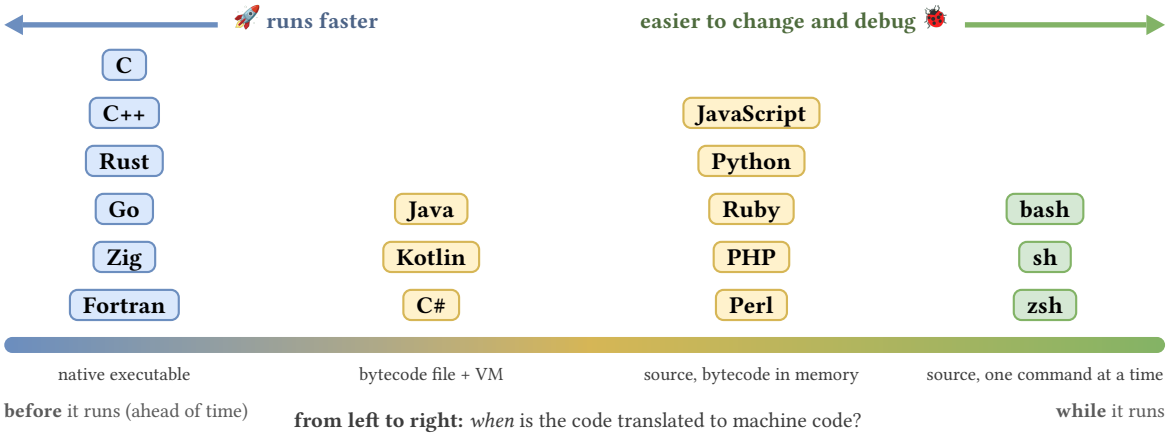


💡 you run the **source**, the interpreter compiles it to bytecode when it starts

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

The Big Picture

The line between *compiled* and *interpreted* is blurry

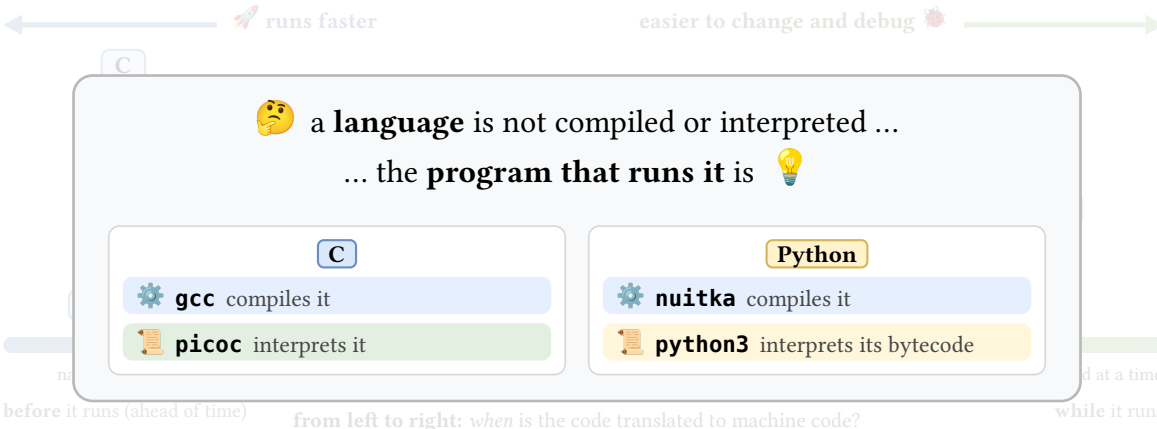


💡 shells really read and run one command at a time

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

The Big Picture

The line between *compiled* and *interpreted* is blurry



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

Compiled vs Interpreted

Neither is better: it depends on what you need

	Compiled (C, Rust, Go)	Interpreted (bash, Python, JS)
When is the code translated?	once, before it runs	every time it runs
How fast does it run?	✅ fast	❌ slower (a JIT helps)
When do you find errors?	✅ syntax and types: before it runs	❌ many only when that line runs
You changed the code?	❌ compile again, then run	✅ run it right away
Another CPU or OS?	❌ compile again for it	✅ runs if it has the interpreter
What do you give to users?	the executable	the source (or bytecode)
Good for	OS, drivers, games, tools	scripts, web, data science

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

Compiled vs Interpreted

Neither is better: it depends on what you need

	Compiled (C, Rust, Go)	Interpreted (bash, Python, JS)
When is the code translated?	once, before it runs	every time it runs
How fast does it run?	✅ fast	❌ slower (a JIT helps)
When do you find errors?	✅ syntax and types: before it runs	❌ many only when that line runs
You changed the code?	❌ compile again, then run	✅ run it right away
Another CPU or OS?	❌ compile again for it	✅ runs if it has the interpreter
What do you give to users?	the executable	the source (or bytecode)
Good for	OS, drivers, games, tools	scripts, web, data science

🚀 compiled: **faster to run**

✍️ interpreted: **faster to write and change**

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

Integrated Development Environments

All the tools in one application

Bibliography

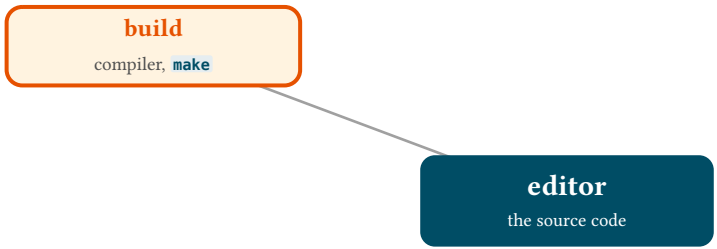
For this section

1. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.6.4 - *Medii integrate de dezvoltare (IDE)*
     - Section 6.8 - *Depanarea programelor*

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program



💡 the IDE runs the build (compiler, `make`, ...) and shows the errors in the code

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program



💡 build and run with one key, the output is in a panel of the IDE

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program



💡 the IDE **understands** the code: the type of a variable, where a function is defined, ...

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program

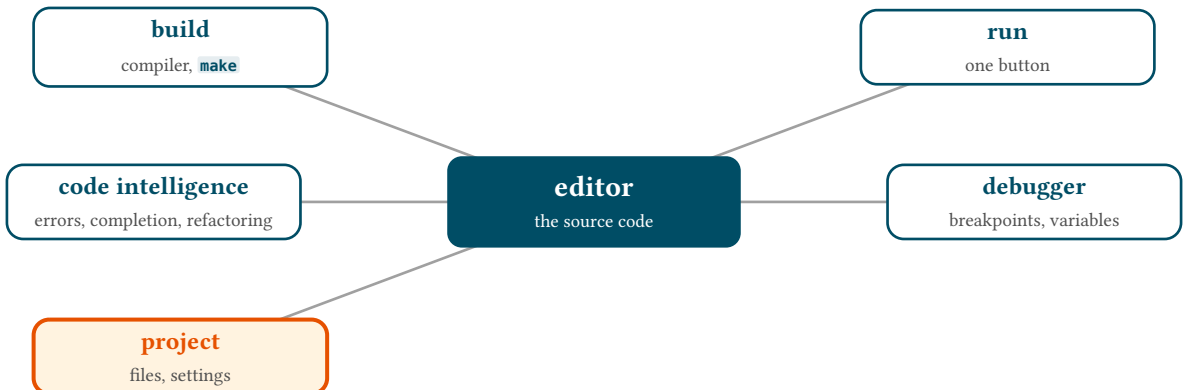


💡 stop the program on a line and look at its variables, see the debugger section

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program

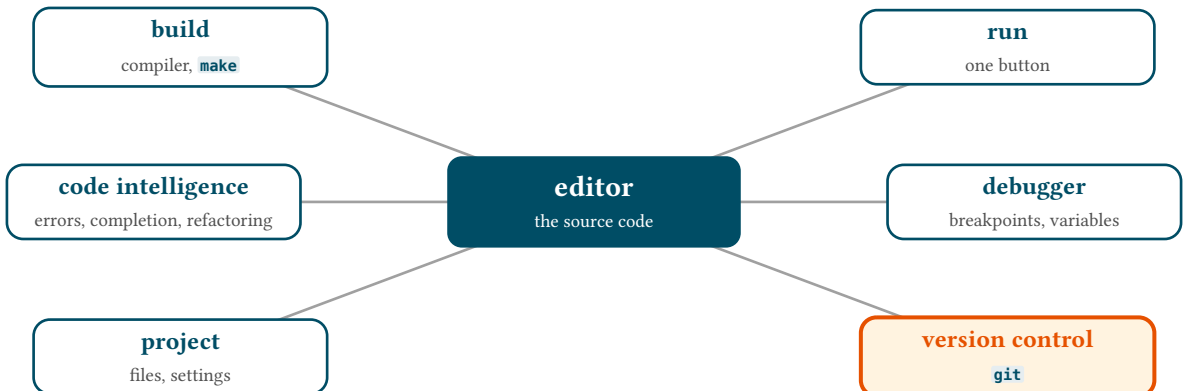


💡 a project: the files of the program and how to build them

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program

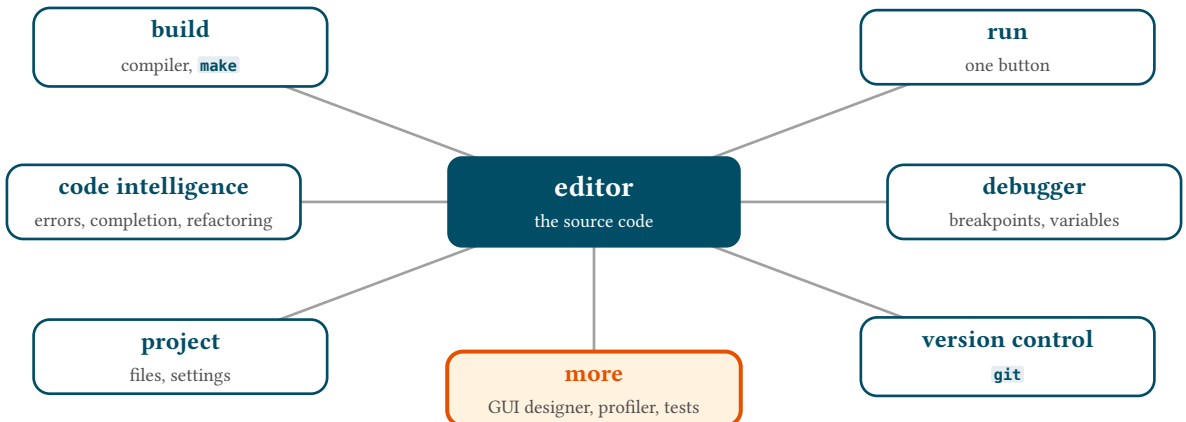


💡 see the changes, commit, push, without leaving the IDE

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

What is an IDE?

Integrated Development Environment: the editor and the development tools, in one program



💡 an editor edits text, an IDE knows **the language** and **the project**

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

Examples of IDEs

Usually made for one language or one platform

Visual Studio — Microsoft
C++ and C# on Windows

CLion, PyCharm, RustRover — JetBrains
C/C++, Python, Rust

Xcode — Apple
macOS and iOS apps, Swift

Qt Creator — Qt
C++ with the Qt library

IntelliJ IDEA — JetBrains
Java, Kotlin

Android Studio — Google
Android apps

Eclipse — open source
Java, C/C++

Code::Blocks — open source
C/C++, small

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

Editor or IDE?

An editor edits text, an IDE knows the language and the project

	Text editor	IDE
understands	text	the language, the project
languages	any	one or a few
build, run, debug	in a terminal	buttons and panels
resources	small, starts instantly	large, slow to start
examples	`vim`, `nano`, `emacs`	Visual Studio, IntelliJ IDEA

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

Editor or IDE?

An editor edits text, an IDE knows the language and the project

	Text editor	IDE
understands	text	the language, the project
languages	any	one or a few
build, run, debug	in a terminal	buttons and panels
resources	small, starts instantly	large, slow to start
examples	`vim`, `nano`, `emacs`	Visual Studio, IntelliJ IDEA

an editor + **plugins** = an IDE. How? 👉 **language servers** and **debug adapters**

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

Language Servers

One program that understands a language, for every editor

Bibliography

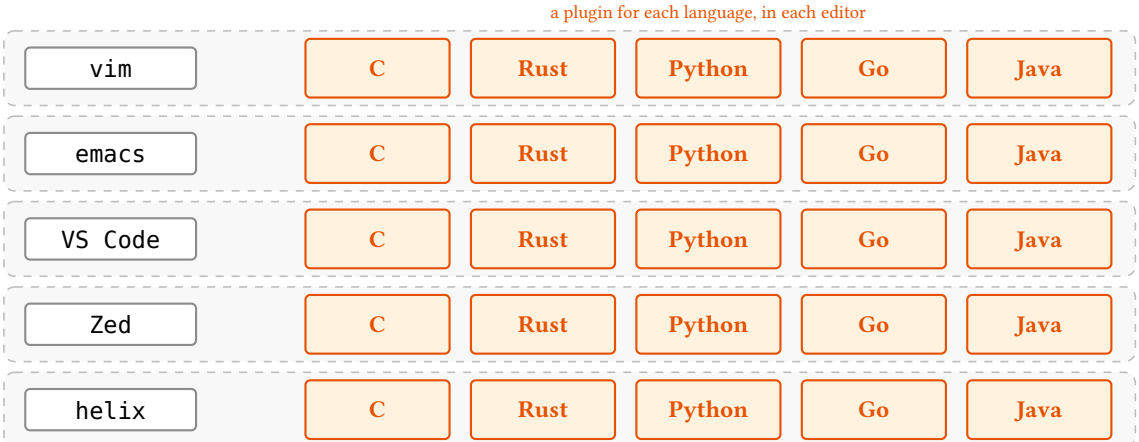
For this section

1. *Language Server Protocol Specification*, online
   - *Overview*
2. *clangd documentation*, online
   - *Getting started* (`compile_commands.json`)

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

The Problem

Every editor wants to understand every language



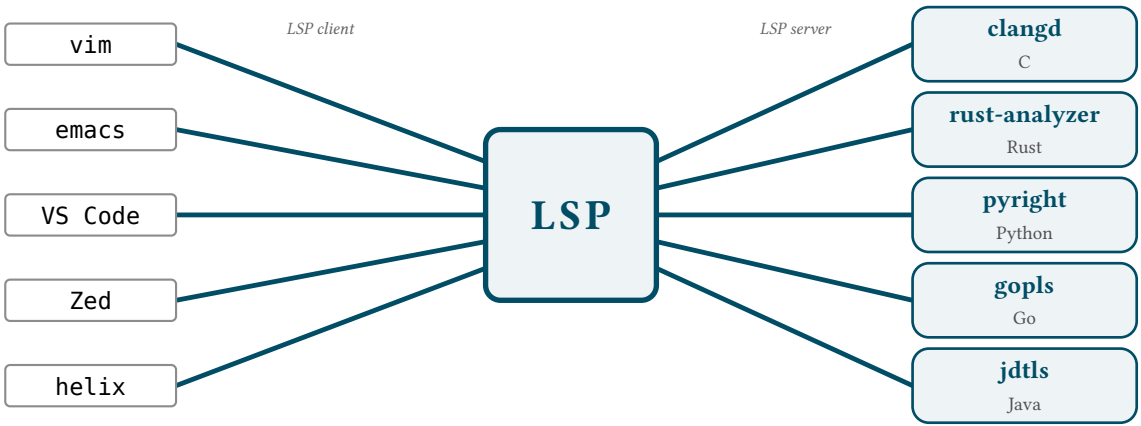
5 × 5 = 25 plugins, each one writes its own C parser, Rust parser, …

💡 understanding C means: parse it, follow the `#include`s, know the types, … a compiler front end

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

The Problem

Every editor wants to understand every language



5 + 5 = 10: every editor and every server talk through the same interface

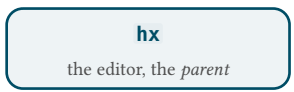
the *Language Server Protocol* (LSP, Microsoft, 2016): the editor asks, the **language server** answers

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

What is a Language Server?

A separate program that understands one language: a child process of the editor

```
$ hx main.c
```



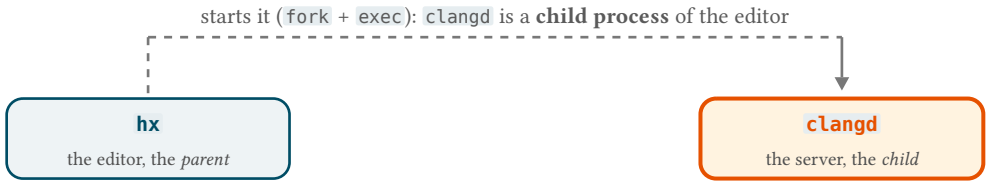
How it starts

1. you run `hx main.c`
2.
3.
4.
5.

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

What is a Language Server?

A separate program that understands one language: a child process of the editor



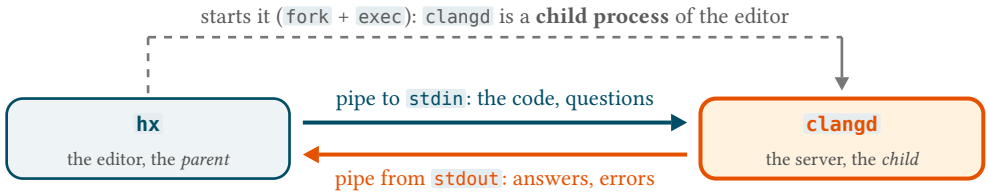
How it starts

1. you run `hx main.c`
2. `hx` sees a C file and **starts** `clangd`
3.
4.
5.

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

What is a Language Server?

A separate program that understands one language: a child process of the editor



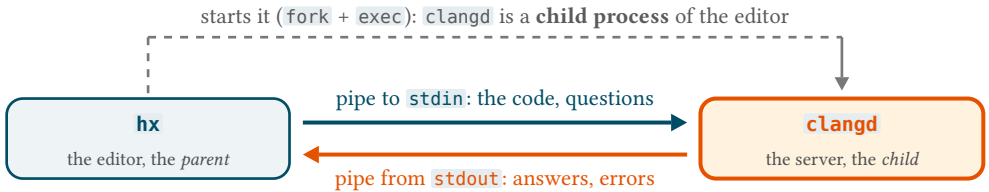
How it starts

1. you run `hx main.c`
2. `hx` sees a C file and **starts** `clangd`
3. two **pipes** connect them
4.
5.

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

What is a Language Server?

A separate program that understands one language: a child process of the editor



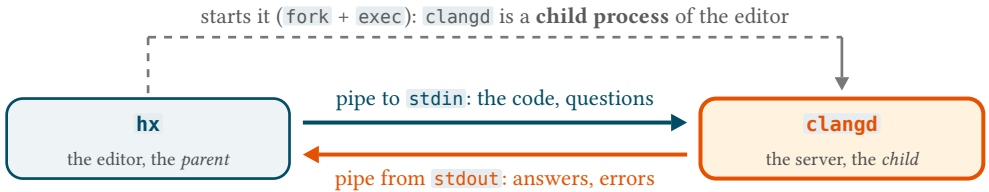
How it starts

1. you run `hx main.c`
2. `hx` sees a C file and **starts** `clangd`
3. two **pipes** connect them
4. `hx` sends the code as you type, and **shows** the answers
5. `clangd` crashes? `hx` keeps working

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

What is a Language Server?

A separate program that understands one language: a child process of the editor



How it starts

1. you run `hx main.c`
2. `hx` sees a C file and **starts** `clangd`
3. two **pipes** connect them
4. `hx` sends the code as you type, and **shows** the answers
5. `clangd` crashes? `hx` keeps working

The server knows

- **diagnostics**, while you type
- **completion**, **hover**
- **go to definition**, **references**
- **rename**, **format**

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

A Conversation with `clangd`



Notifications and requests

main.c

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

NOR 1:1

Notifications: something happened, no answer

editor ⟶ clangd: textDocument/didOpen notification

```
uri:  file:///home/student/project/main.c
text: "#include \"lib.h\"\n#include ..."  (the whole file)
```

💡 the editor opened `main.c`: it sends the whole text (and `didChange` after every edit)

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

A Conversation with `clangd`

Notifications and requests

main.c

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

NOR 1:1

Notifications: something happened, no answer

editor → clangd: textDocument/didOpen notification

```
uri:  file:///home/student/project/main.c
text: "#include \"lib.h\"\n#include ..."  (the whole file)
```

clangd → editor: textDocument/publishDiagnostics

notification

```
uri:     file:///home/student/project/main.c
range:   line 6, characters 24 - 28
message: "Use of undeclared identifier 'totl';
          did you mean 'total'?"
```

💡 when it finishes checking, the server sends the errors: the editor underlines `totl`

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

A Conversation with `clangd`



Notifications and requests

main.c

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

NOR 6:17

the cursor: line 6, column 17
LSP counts from 0: `line 5`, `character 16`

Requests: a question and its answer, the same `id`

editor → `clangd`: `textDocument/hover` request

```
id:       2
uri:      file:///home/student/project/main.c
position: line 5, character 16
```

💡 the cursor is on `sum` and you press `Space` `k` in `helix`: the editor asks what is there

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

A Conversation with `clangd`



Notifications and requests

main.c

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

NOR 6:17

the cursor: line 6, column 17
LSP counts from 0: `line 5`, `character 16`

Requests: a question and its answer, the same `id`

editor → clangd: textDocument/hover request

```
id:       2
uri:      file:///home/student/project/main.c
position: line 5, character 16
```

clangd → editor: the answer to request 2 answer

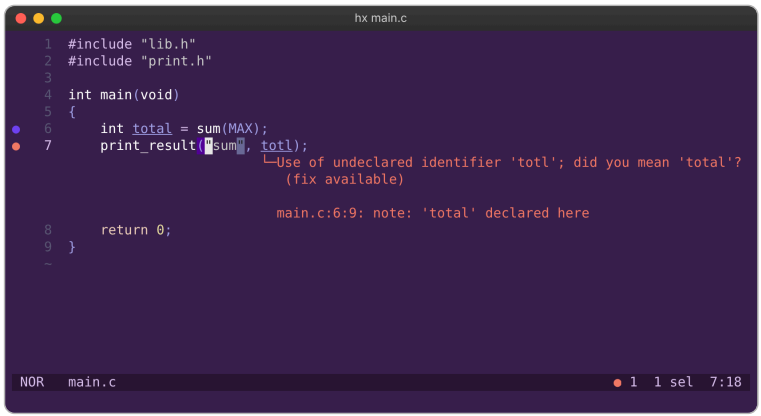
```
id: 2
function sum -> int
int sum(int n), provided by "lib.h"
```

💡 the editor shows the answer in a popup

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

What You Get in `helix`

`clangd` in `helix` (built in) and in VSCodium (an extension)



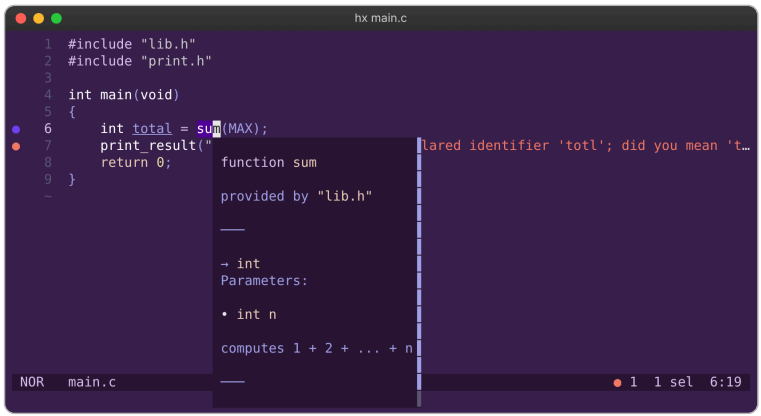
Diagnostics

- the red dot: an error on line 7
- the message and the fix, while you type, before compiling

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

What You Get in `helix`

`clangd` in `helix` (built in) and in VSCodium (an extension)



Hover

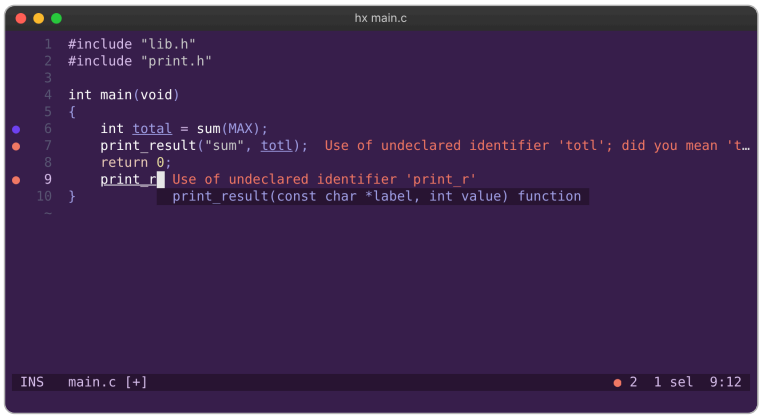
Space k on `sum`:

- its type and parameters
- the header it comes from
- its comment from `lib.h`

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

What You Get in `helix`

`clangd` in `helix` (built in) and in VSCodium (an extension)



Completion

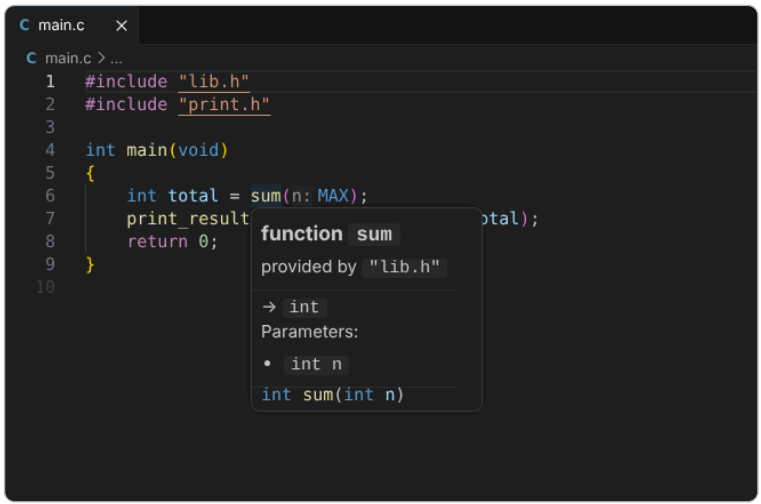
typing `print_r`:

- the server knows `print_result` from `print.h`
- with its parameters

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

What You Get in VSCodium

`clangd` in `helix` (built in) and in VSCodium (an extension)



The same server

- VSCodium with the `clangd` extension: the same hover, from the same `clangd`
- the gray `n:` is an *inlay hint*: the name of the parameter

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

Language Servers

Install the server, the editor finds it in `PATH`

language	server
C, C++	`clangd`
Rust	`rust-analyzer`
Python	`pyright`, `basedpyright`, `pylsp`
Go	`gopls`
Java	`jdtls`
JavaScript, TypeScript	`typescript-language-server`
`bash`	`bash-language-server`
Typst	`tinymist`

```
$ hx --health c
Configured language servers:
  ✓ clangd: /usr/bin/clangd
```

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

Language Servers

Install the server, the editor finds it in `PATH`

language	server
C, C++	`clangd`
Rust	`rust-analyzer`
Python	`pyright`, `basedpyright`, `pylsp`
Go	`gopls`
Java	`jdtls`
JavaScript, TypeScript	`typescript-language-server`
`bash`	`bash-language-server`
Typst	`tinymist`

```
$ hx --health c
Configured language servers:
  ✓ clangd: /usr/bin/clangd
```

⚠ clangd and the compiler flags

`clangd` needs the flags (`-I`, `-D`, ...) from `compile_commands.json`

```
$ bear -- make
```

`bear` watches the build and writes it

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

Debuggers

Stop the program and look inside

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 6.8 - *Depanarea programelor*
2. *Debugging with GDB*, online

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

What is a Debugger?

A program that runs **another program** under its control

Instead of `printf`

- **breakpoints**: stop at a line or a function
- **step**: run one line at a time
- **inspect**: variables, memory, the call stack
- **watchpoints**: stop when a variable changes
- **change** a variable and continue

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

What is a Debugger?

A program that runs **another program** under its control

Instead of `printf`

- **breakpoints**: stop at a line or a function
- **step**: run one line at a time
- **inspect**: variables, memory, the call stack
- **watchpoints**: stop when a variable changes
- **change** a variable and continue

Debuggers

- `gdb`, the GNU debugger
- `lldb`, from LLVM (the `clang` project)
- the Visual Studio debugger, on Windows

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

How Does a Debugger Work?



🐧 On Linux, the debugger is the **parent** of the program and controls it with `ptrace`

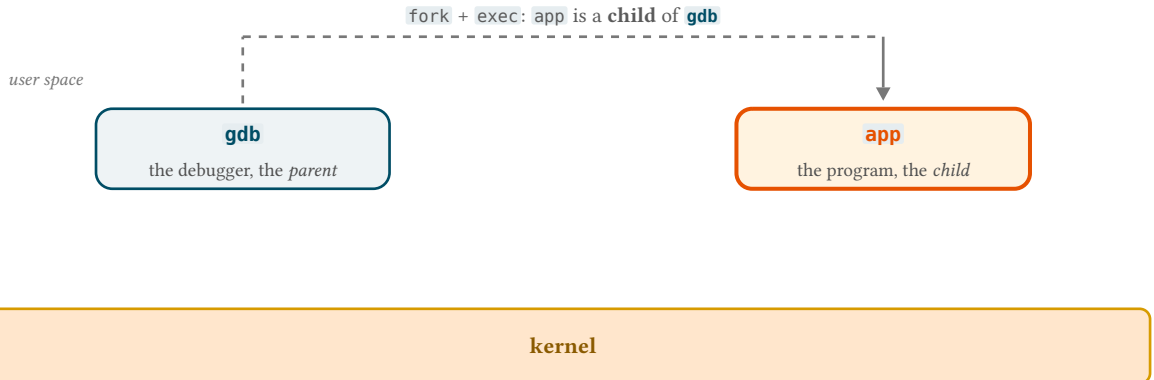


💡 `gdb` starts the program itself

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

How Does a Debugger Work?

🐧 On Linux, the debugger is the **parent** of the program and controls it with `ptrace`

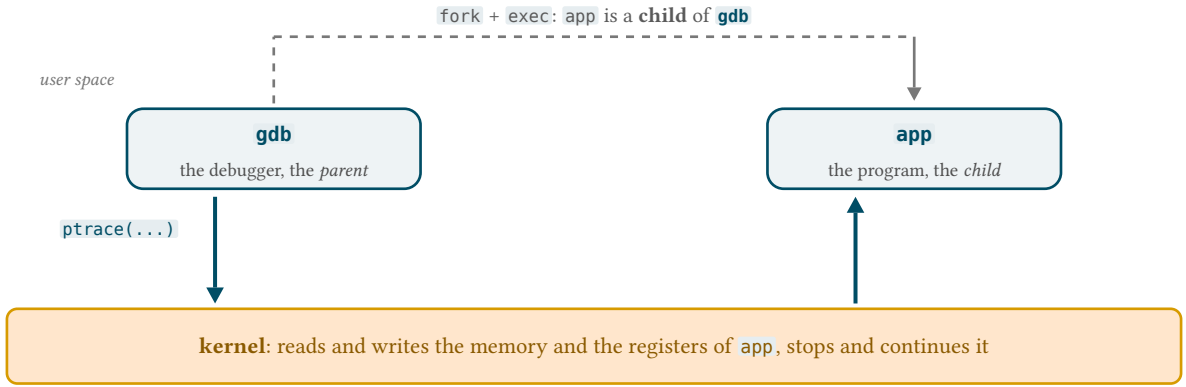


💡 `fork` + `exec`: `app` is a **child** of `gdb`, stopped before its first instruction

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

How Does a Debugger Work?

🐧 On Linux, the debugger is the **parent** of the program and controls it with `ptrace`

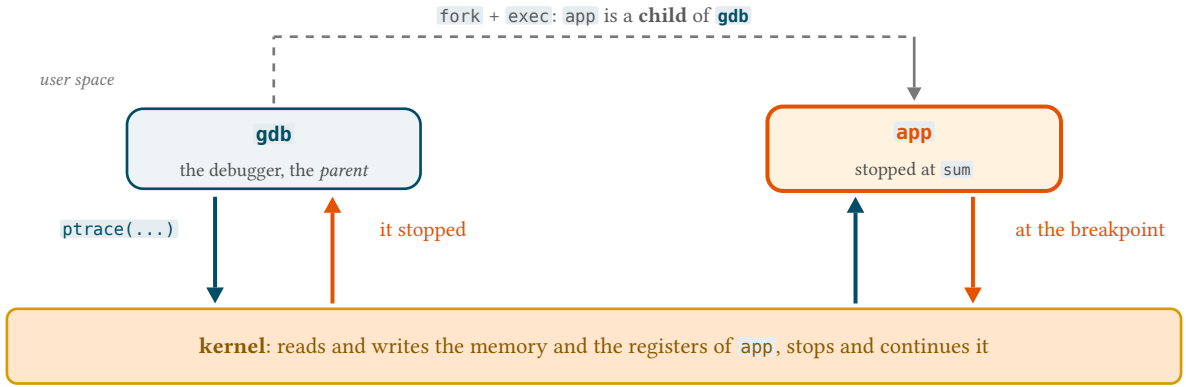


💡 the `ptrace` system call: `gdb` asks the kernel to read and write `app`, to stop it and continue it

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

How Does a Debugger Work?

🐧 On Linux, the debugger is the **parent** of the program and controls it with `ptrace`



💡 `app` reaches the breakpoint: the kernel stops it and tells `gdb`

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

A `gdb` Session

`app` is built with `-g`

lib.c

```c
int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

```
$ gdb -q ./app
Reading symbols from ./app...
(gdb)
```

💡 start the program **under** `gdb`, it does not run yet

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

A `gdb` Session

`app` is built with `-g`

lib.c

```
1  int sum(int n)
2  {
3  ▶    int s = 0;
4      for (int i = 1; i <= n; i++) {
5          s = s + i;
6      }
7      return s;
8  }
```

```
$ gdb -q ./app
Reading symbols from ./app...
(gdb) break sum
Breakpoint 1 at 0x1188: file lib.c, line 3.
(gdb)
```

💡 `break`: a **breakpoint** at the function `sum`, i.e. at the first line of its body, line 3

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

A `gdb` Session

`app` is built with `-g`

lib.c

```
1  int sum(int n)
2  {
3  ▶       int s = 0;
4          for (int i = 1; i <= n; i++) {
5              s = s + i;
6          }
7          return s;
8  }
```

```
$ gdb -q ./app
Reading symbols from ./app...
(gdb) break sum
Breakpoint 1 at 0x1188: file lib.c, line 3.
(gdb) run
Starting program: /home/student/project/app
Breakpoint 1, sum (n=10) at lib.c:3
3           int s = 0;
(gdb)
```

💡 `run`: the program starts and stops at the breakpoint, **before** line 3 runs

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

A `gdb` Session

`app` is built with `-g`

lib.c

```
1 ▶ int sum(int n)
2   {
3 ▶     int s = 0;
4       for (int i = 1; i <= n; i++) {
5           s = s + i;
6       }
7       return s;
8   }
```

```
Reading symbols from ./app...
(gdb) break sum
Breakpoint 1 at 0x1188: file lib.c, line 3.
(gdb) run
Starting program: /home/student/project/app
Breakpoint 1, sum (n=10) at lib.c:3
3           int s = 0;
(gdb) print n
$1 = 10
(gdb)
```

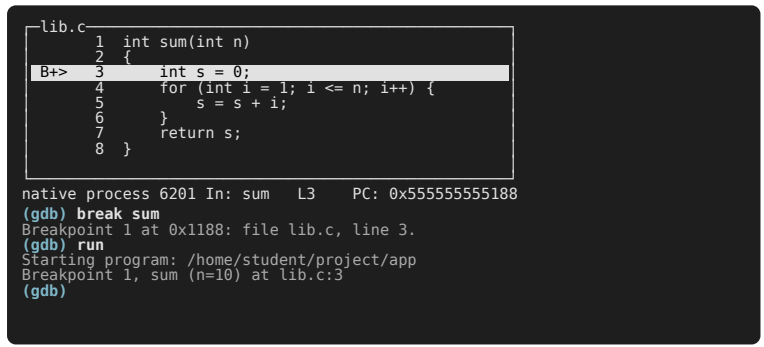
💡 `print`: the value of a variable, or of any C expression (`n * 2`)

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

`gdb` with a Text Interface

`gdb -tui`: the code above, the commands below

- start it with `gdb -tui ./app`, or press `Ctrl` + `X` `A` in `gdb`
- the **source window** follows the program
  - `B+` a breakpoint
  - `>` the next line to run
- the **command window**: the same commands as before



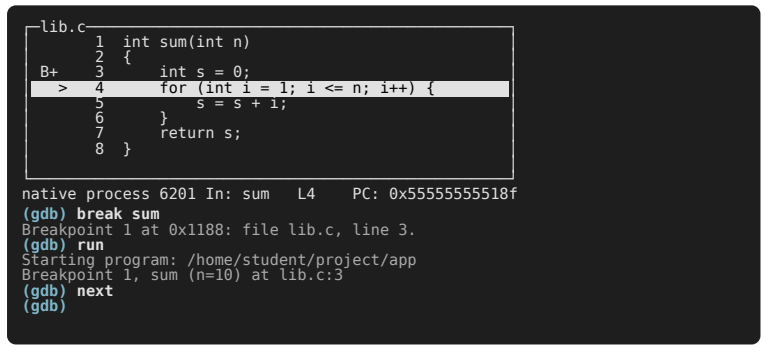
💡 stopped at the breakpoint: the source window shows **where**

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

`gdb` with a Text Interface

`gdb -tui`: the code above, the commands below

- start it with `gdb -tui ./app`, or press `Ctrl` + `X` `A` in `gdb`
- the **source window** follows the program
  - `B+` a breakpoint
  - `>` the next line to run
- the **command window**: the same commands as before



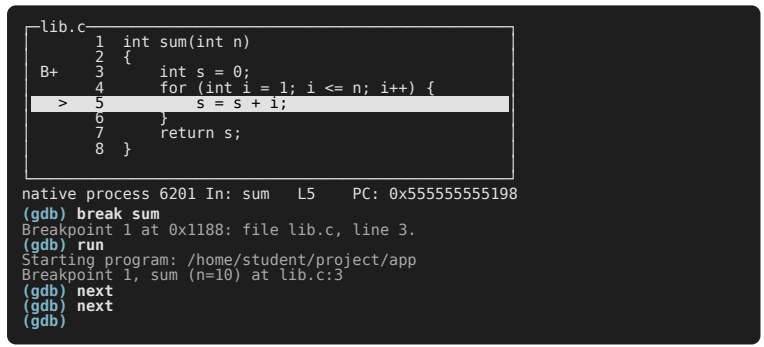
💡 `next`: the `>` moves, no need to print the line in the command window

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

`gdb` with a Text Interface

`gdb -tui`: the code above, the commands below

- start it with `gdb -tui ./app`, or press `Ctrl` + `X` `A` in `gdb`
- the **source window** follows the program
  - `B+` a breakpoint
  - `>` the next line to run
- the **command window**: the same commands as before

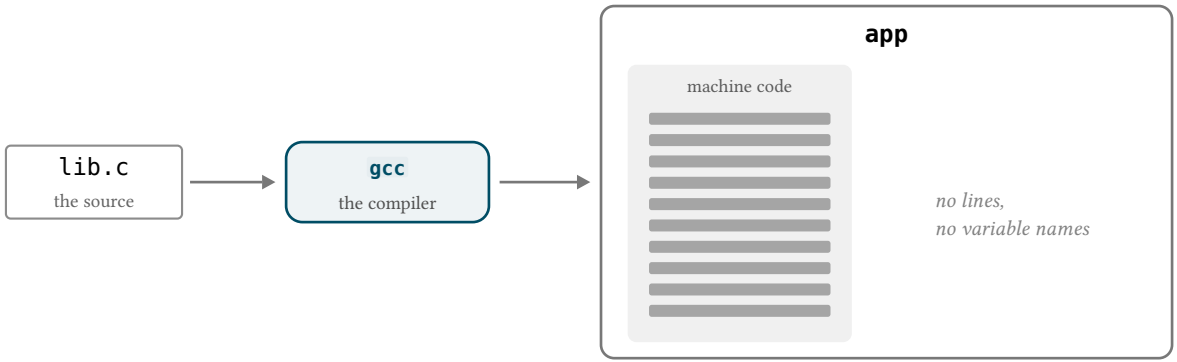


💡 `next`: the `>` moves, no need to print the line in the command window

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

Debug Information

What the debugger needs, and `-g` adds

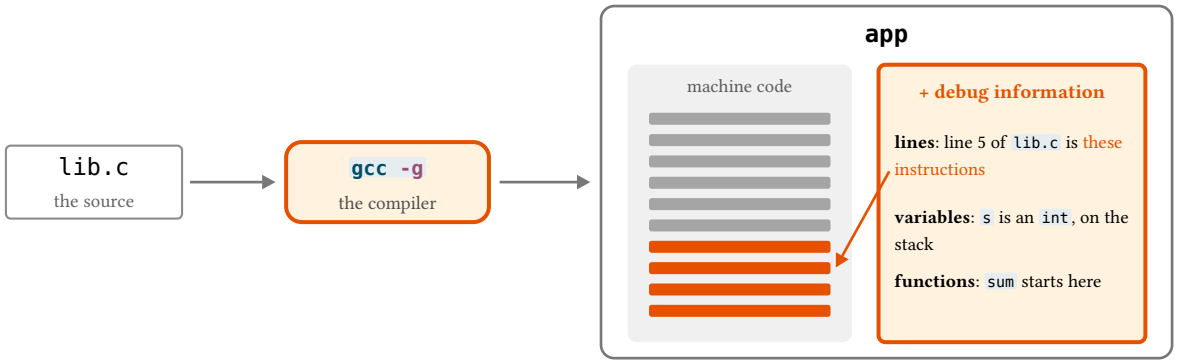


💡 without it, `app` is only machine code: the debugger cannot show lines or variables

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

Debug Information

What the debugger needs, and `-g` adds

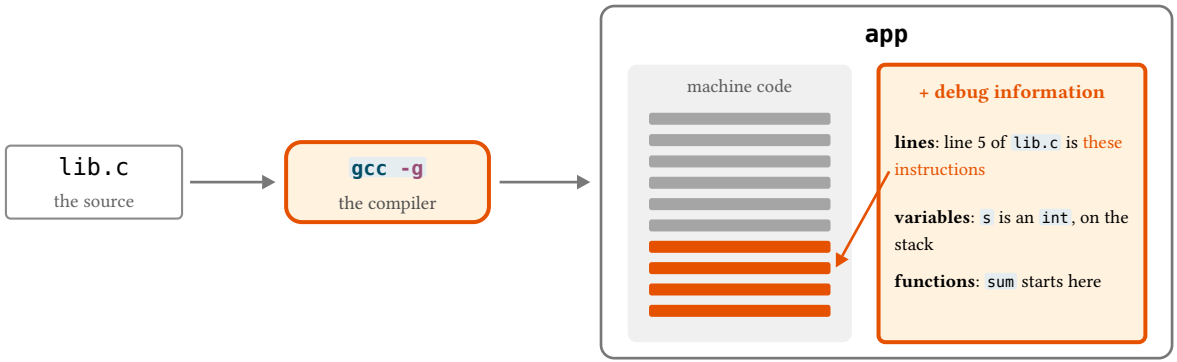


💡 `-g` adds a map from the machine code back to the source

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

Debug Information

What the debugger needs, and `-g` adds



C `gcc -g` **Rust** `cargo build` **Go** `go build` **Java** `javac -g`

💡 a *debug* build adds it, a *release* build usually does not

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

Debug Adapters

One program that drives a debugger, for every editor

Bibliography

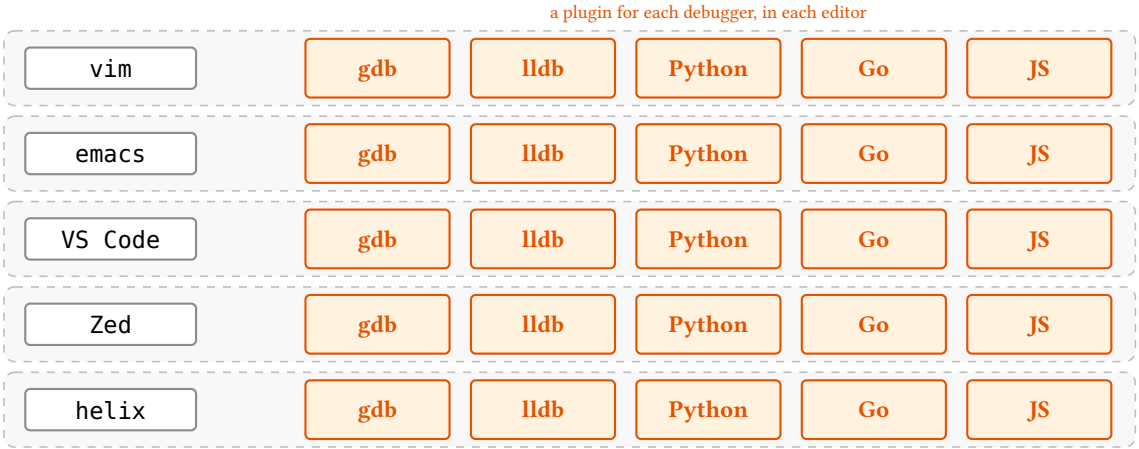
For this section

1. *Debug Adapter Protocol Specification*, online
   - *Overview*
2. *Debugging with GDB*, online
   - *Debugger Adapter Protocol*

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

The Same Problem

Every editor wants to drive every debugger

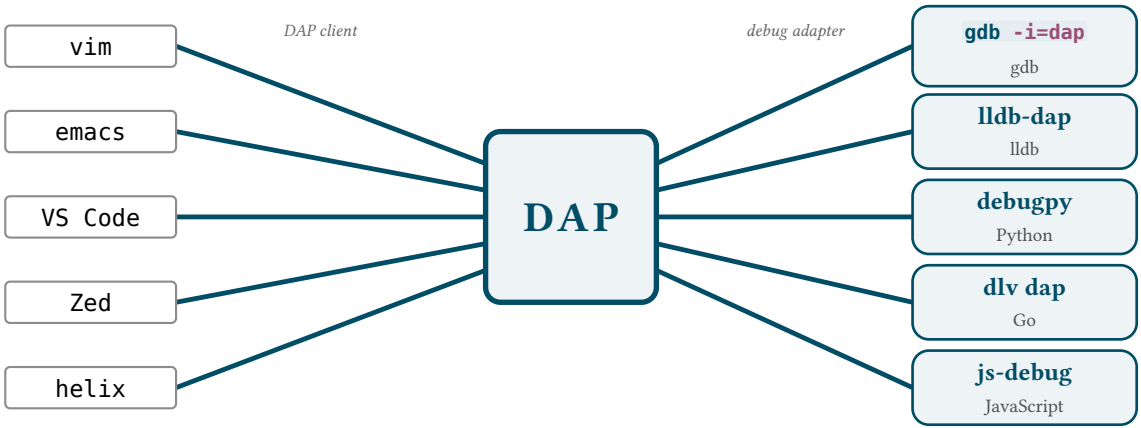


💡 every debugger has its own commands: `break` in **`gdb`**, `breakpoint set` in **`lldb`**

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

The Same Problem

Every editor wants to drive every debugger



5 + 5 = 10: every editor and every debugger talk through the same interface

the *Debug Adapter Protocol* (DAP, Microsoft, 2016): the editor asks, the **debug adapter** answers

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

What is a Debug Adapter?

A program that speaks DAP and drives a debugger: a child process of the editor



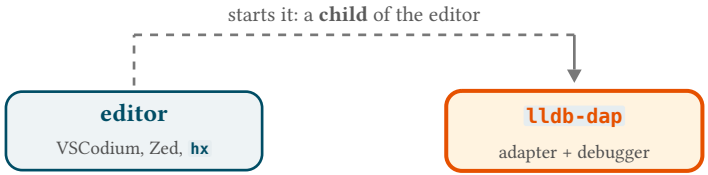
How it starts

1. you start debugging in the editor
2.
3.
4.

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

What is a Debug Adapter?

A program that speaks DAP and drives a debugger: a child process of the editor



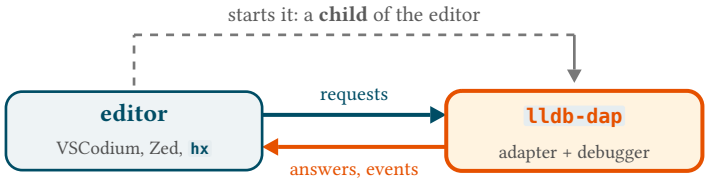
How it starts

1. you start debugging in the editor
2. the editor **starts** the adapter, e.g. `lldb-dap`
3.
4.

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

What is a Debug Adapter?

A program that speaks DAP and drives a debugger: a child process of the editor



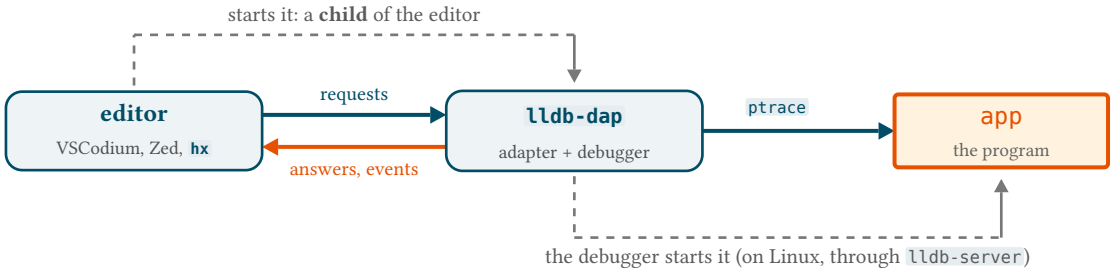
How it starts

1. you start debugging in the editor
2. the editor **starts** the adapter, e.g. `lldb-dap`
3. two **pipes** connect them, as for LSP
4.

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

What is a Debug Adapter?

A program that speaks DAP and drives a debugger: a child process of the editor



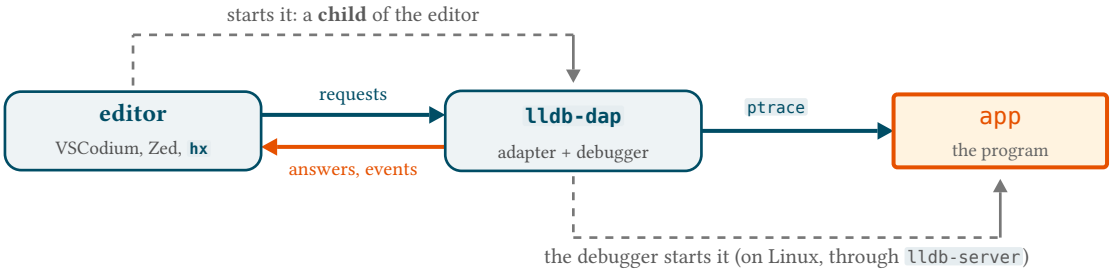
How it starts

1. you start debugging in the editor
2. the editor **starts** the adapter, e.g. `lldb-dap`
3. two **pipes** connect them, as for LSP
4. its debugger **starts** `app` and controls it with `ptrace` (on Linux, `lldb` uses a helper, `lldb-server`)

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

What is a Debug Adapter?

A program that speaks DAP and drives a debugger: a child process of the editor



How it starts

1. you start debugging in the editor
2. the editor **starts** the adapter, e.g. `lldb-dap`
3. two **pipes** connect them, as for LSP
4. its debugger **starts** `app` and controls it with `ptrace` (on Linux, `lldb` uses a helper, `lldb-server`)

The adapter knows

- **breakpoints**
- **continue**, **step**
- the **call stack**
- the **variables**
- **evaluate** an expression

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

A Conversation with `lldb-dap`

Requests, responses and events

lib.c

```
1 int sum(int n)
2 {
3     int s = 0;
4     for (int i = 1; i <= n; i++) {
5         s = s + i;
6     }
7     return s;
8 }
```

app: running

Requests: a question and its answer

editor → adapter: `setBreakpoints` request

```
seq:         3
source:      /home/student/project/lib.c
breakpoints: [ line 5 ]
```

💡 you click next to line 5: the editor asks for a breakpoint (DAP counts lines from 1)

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

A Conversation with `lldb-dap`

Requests, responses and events

lib.c

```
1 int sum(int n)
2 {
3     int s = 0;
4     for (int i = 1; i <= n; i++) {
5         s = s + i;
6     }
7     return s;
8 }
```

app: running

Requests: a question and its answer

editor → adapter: `setBreakpoints` request

```
seq:         3
source:      /home/student/project/lib.c
breakpoints: [ line 5 ]
```

adapter → editor: the answer to request 3 response

```
request_seq: 3
success:     true
breakpoints: [ line 5, verified ]
```

💡 the adapter set it in `lldb`: the breakpoint is *verified*, the dot turns red

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

A Conversation with `lldb-dap`



Requests, responses and events

lib.c

```
1 int sum(int n)
2 {
3     int s = 0;
4     for (int i = 1; i <= n; i++) {
5         s = s + i;
6     }
7     return s;
8 }
```

app: **paused**

Events: something happened, no answer

adapter → editor: `stopped` event

```
reason:   breakpoint
threadId: 1
```

💡 `app` reached line 5: the adapter tells the editor, which highlights the line

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

A Conversation with `lldb-dap`

Requests, responses and events

lib.c

```
1 int sum(int n)
2 {
3     int s = 0;
4     for (int i = 1; i <= n; i++) {
5         s = s + i;
6     }
7     return s;
8 }
```

app: **paused**

Requests: a question and its answer

editor → adapter: `variables` request

```
seq:                9
variablesReference: 1001
```

💡 the editor asks for the local variables of `sum` (`1001` came from earlier requests)

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

A Conversation with `lldb-dap`

Requests, responses and events

lib.c

```
1 int sum(int n)
2 {
3     int s = 0;
4     for (int i = 1; i <= n; i++) {
5         s = s + i;
6     }
7     return s;
8 }
```

app: **paused**

Variables: `n = 10`, `s = 0`, `i = 1`

Requests: a question and its answer

editor → adapter: `variables` request

```
seq:                9
variablesReference: 1001
```

adapter → editor: `the answer to request 9` response

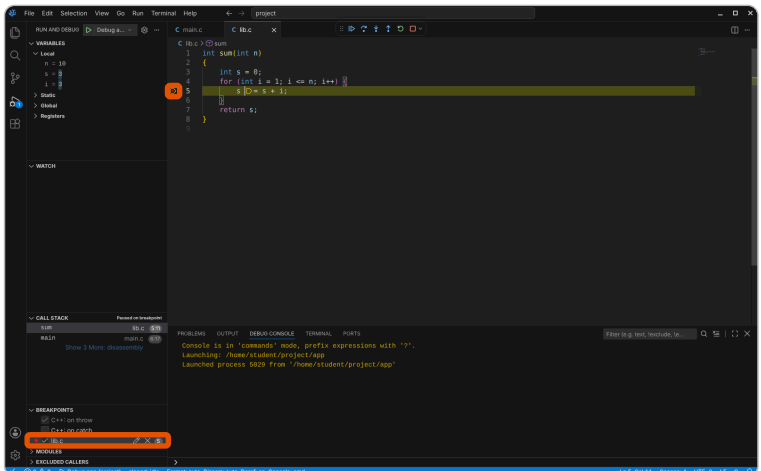
```
request_seq: 9
variables:   n = 10   (int)
             s = 0    (int)
             i = 1    (int)
```

💡 the editor shows them in the *Variables* panel

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

What You Get in VSCodium

`lldb` in VSCodium (CodeLLDB) and in `nvim` (`nvim-dap`)



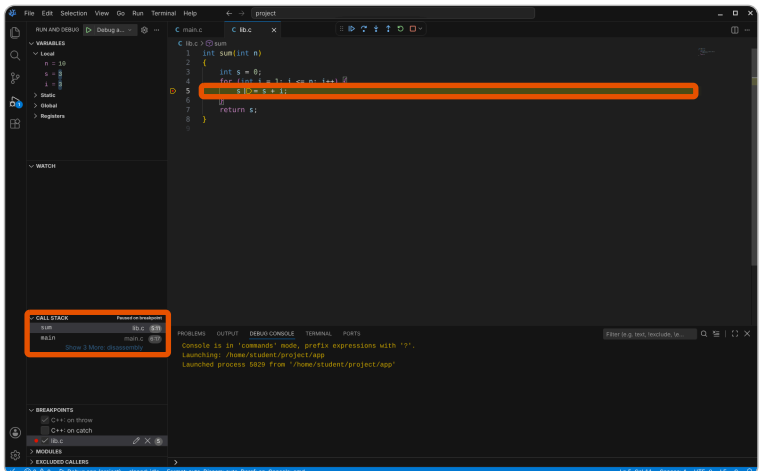
Breakpoints

- click next to a line: `setBreakpoints`
- the red dot on line 5, the list at the bottom left

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

What You Get in VSCodium

`lldb` in VSCodium (CodeLLDB) and in `nvim` (`nvim-dap`)



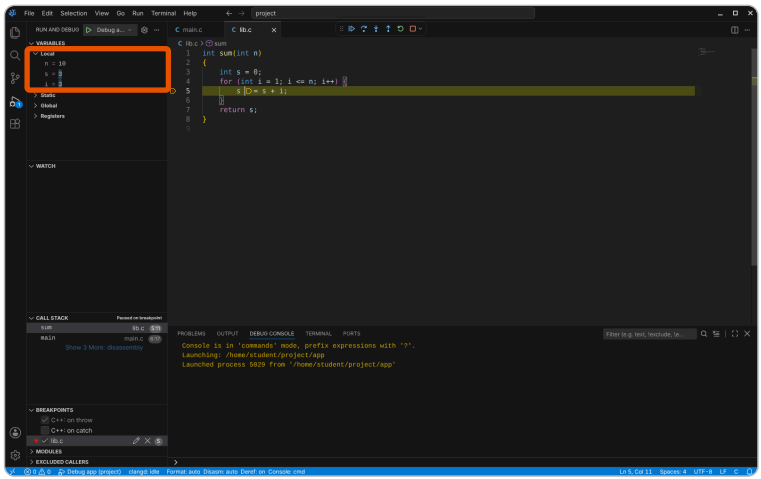
Stopped

- the `stopped` event: line 5 is highlighted
- `stackTrace`: the *Call Stack*, `sum` called by `main`

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

What You Get in VSCodium

`lldb` in VSCodium (CodeLLDB) and in `nvim` (`nvim-dap`)



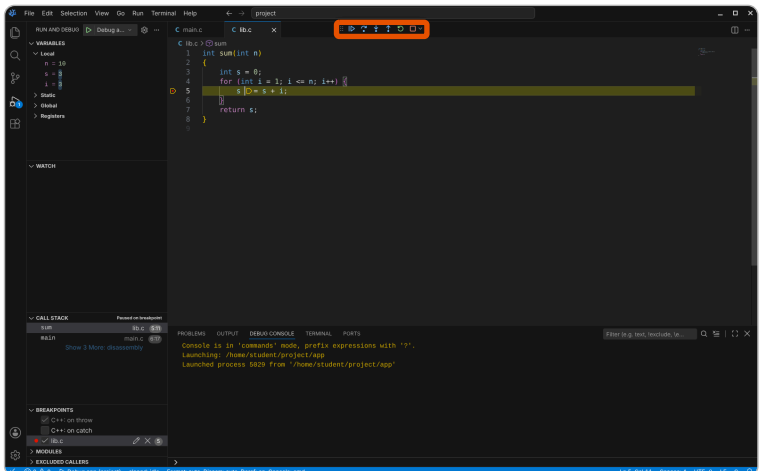
Variables

- `scopes` + `variables`: the *Variables* panel
- `n`, `s`, `i` of `sum`, here in the 3rd iteration

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

What You Get in VSCodium

`lldb` in VSCodium (CodeLLDB) and in `nvim` (`nvim-dap`)



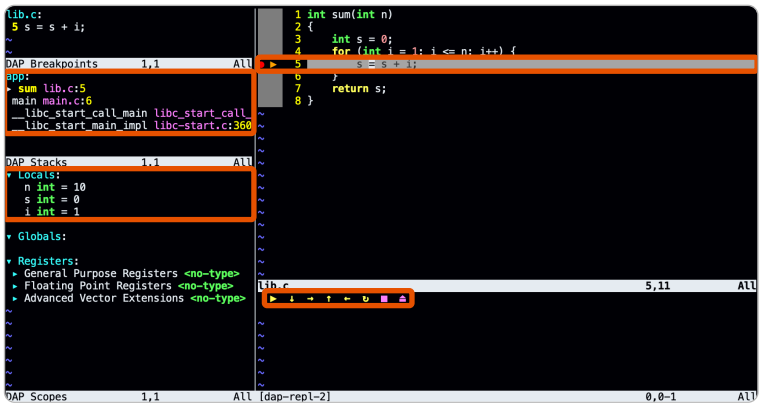
Buttons

- the toolbar at the top: continue, step over, step into, step out, restart, stop
- one DAP request for each button

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

What You Get in `nvim`

`lldb` in VSCodium (CodeLLDB) and in `nvim` (`nvim-dap`)



The same protocol, in a terminal

- `nvim` + the `nvim-dap` plugin + `lldb-dap`
- the same answers: the breakpoint, the stopped line, the call stack, the variables, the buttons

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

Debug Adapters

Install the adapter, the editor starts it

language	debug adapter
C, C++, Rust	`lldb-dap`, CodeLLDB, `gdb -i=dap`[1]
Python	`debugpy`
Go	`dlv dap` (Delve)
JavaScript, TypeScript	`js-debug`

⚠ the adapter needs the **debug information**: compile with `-g`

```
$ gcc -g main.c lib.c print.c -o app
```

⚠ the editor must know **what** to start: the path of `app`, its arguments, … a *launch configuration*

[1] `gdb` speaks DAP itself since version 14: `-i=dap` (*interpreter*) replaces its command line with DAP

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

The C Compilation Pipeline

From `hello.c` to an executable, in four steps

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.1 - *The C Compiler*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.3 - *Limbaje compilate*
     - Section 6.7.1 - *C / C++*

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

The Example

A small C program, compiled and run

hello.c

```c
#include <stdio.h>

#define TIMES 3

int main(void)
{
    // say hello a few times
    for (int i = 0; i < TIMES; i++) {
        printf("Hello, %d!\n", i);
    }
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
```

- `#include` and `#define`: for the **preprocessor**
- the comment: for us
- `printf` is not written here, it is in the C library (`libc`)

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

The Example

A small C program, compiled and run

hello.c

```c
#include <stdio.h>

#define TIMES 3

int main(void)
{
    // say hello a few times
    for (int i = 0; i < TIMES; i++) {
        printf("Hello, %d!\n", i);
    }
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
```

- `#include` and `#define`: for the **preprocessor**
- the comment: for us
- `printf` is not written here, it is in the C library

Behind the scenes

`gcc hello.c -o hello` does **four** steps, with **three** programs



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

0. The C Source: `hello.c`

One command, four steps



hello.c

```c
#include <stdio.h>

#define TIMES 3

int main(void)
{
    // say hello a few times
    for (int i = 0; i < TIMES; i++) {
        printf("Hello, %d!\n", i);
    }
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, 0!
Hello, 1!
Hello, 2!
```

💡 behind one `gcc` command, **four** steps: `cc1` does the first two, `as` and `ld` the others

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

1. Preprocessor: `gcc -E`

Copies the `#include` files, replaces the `#define`s, removes the comments




```
$ gcc -E hello.c -o hello.i
$ wc -l hello.c hello.i
   12 hello.c
  825 hello.i
$ grep printf hello.i | head -1
extern int printf (const char *__restrict
__format, ...);
```

hello.i (the last lines)

```
# 5 "hello.c"
int main(void)
{

    for (int i = 0; i < 3; i++) {
        printf("Hello, %d!\n", i);
    }
    return 0;
}
```

💡 `stdio.h` is copied in (800 lines, `printf` among them), `TIMES` became `3`, the comment is gone

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

2. Compiler: `gcc -S`

Translates C to **assembly**, the text form of the CPU instructions




```
$ gcc -S hello.c
$ ls
hello.c  hello.s
```

- **assembly**: one line per CPU instruction, still a text file
- for **one** processor: this is x86-64
- the loop became a compare (`cmpl`) and a jump (`jle`)
- `printf` is only **called**, its code is not here

hello.s (excerpt)

```
main:
    ...
    movl    $0, -4(%rbp)
    jmp     .L2
.L3:
    ...
    call    printf@PLT
    addl    $1, -4(%rbp)
.L2:
    cmpl    $2, -4(%rbp)
    jle     .L3
```

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

3. Assembler: `gcc -c`

Translates assembly to **machine code**: an object file




```
$ gcc -c hello.c
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, ...
$ nm hello.o
0000000000000000 T main
                 U printf
```

- an **object file**: machine code, not runnable yet
- `T main`: defined here (in the *text* section)
- `U printf`: used here, **undefined**

```
$ objdump -d hello.o
0000000000000000 <main>:
   4: 55                push %rbp
   5: 48 89 e5          mov  %rsp,%rbp
  ...
  24: b8 00 00 00 00    mov  $0x0,%eax
  29: e8 00 00 00 00    call 2e <main+0x2e>
  2e: 83 45 fc 01       addl $0x1,-0x4(%rbp)
```

💡 `e8` is `call`, to address `00 00 00 00`: the **linker** fills in `printf`

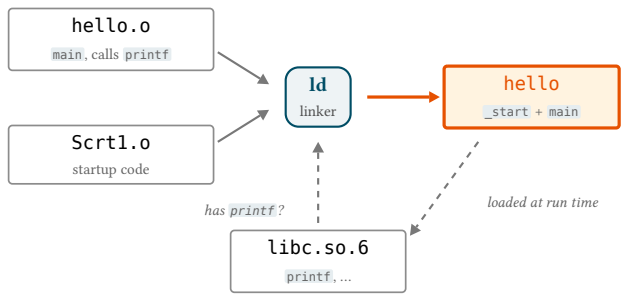
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

4. Linker: `gcc`

Puts the object files and the libraries together, into an executable




```
$ gcc hello.o -o hello
$ file hello
hello: ELF 64-bit ..., dynamically linked, ...
$ ldd hello
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
...
```



💡 `printf` is **not copied**: it stays in `libc.so.6`, a **shared library**

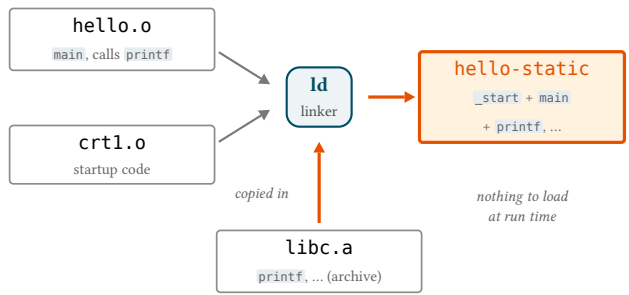
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Static Executable: `gcc -static`

Copies the library code into the executable




```
$ gcc -static hello.o -o hello-static
$ file hello-static
hello-static: ELF 64-bit ..., statically
linked, ...
$ ldd hello-static
not a dynamic executable
$ ls -sh hello hello-static
  12K hello  636K hello-static
```



💡 runs **without** `libc.so.6`, but **every program** carries its own copy of `printf`

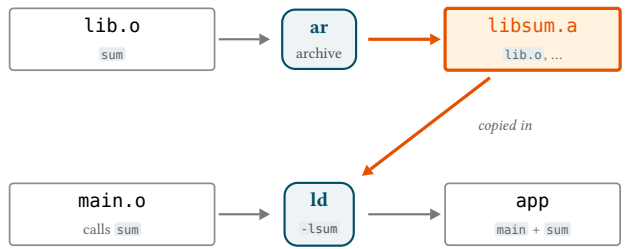
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Static Library: `ar`

Our own `.a`, an archive of object files, copied into every program




```
$ gcc -c lib.c -o lib.o
$ ar rcs libsum.a lib.o
$ ar t libsum.a
lib.o
$ gcc main.o -L. -lsum -o app
$ ./app
sum: 55
```



💡 `app` runs without `libsum.a`; change `lib.c`: rebuild `libsum.a` **and** link `app` again

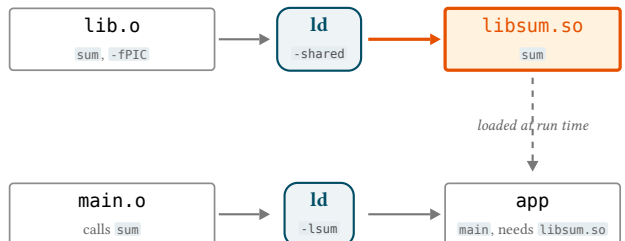
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Shared Library: `gcc -shared`

Our own `.so`, loaded by every program that uses it




```
$ gcc -fPIC -c lib.c -o lib.o
$ gcc -shared lib.o -o libsum.so
$ gcc main.o -L. -lsum -o app
$ ldd app
libsum.so => not found
$ LD_LIBRARY_PATH=. ./app
sum: 55
```



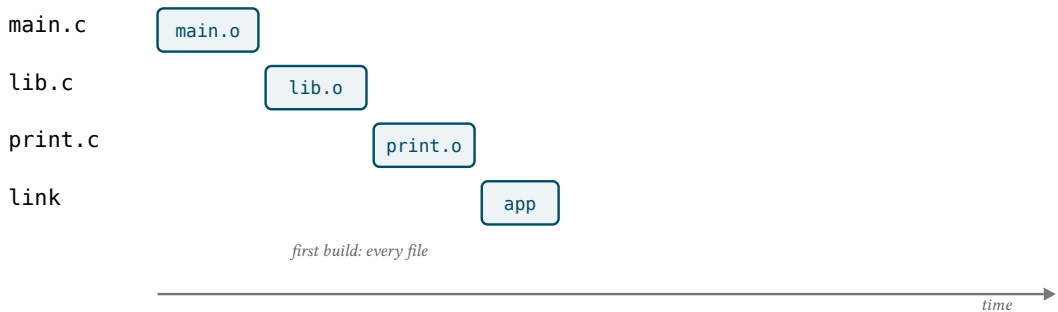
⚠ the loader looks for `libsum.so` only in the system folders (`/usr/lib`, ...): **install** it there, or point to it with `LD_LIBRARY_PATH`

> 💡 change `lib.c`, rebuild `libsum.so`: every program gets the new code **without** linking again

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

Compile Everything, Every Time?

`gcc` with a list of source files compiles **all** of them, every time

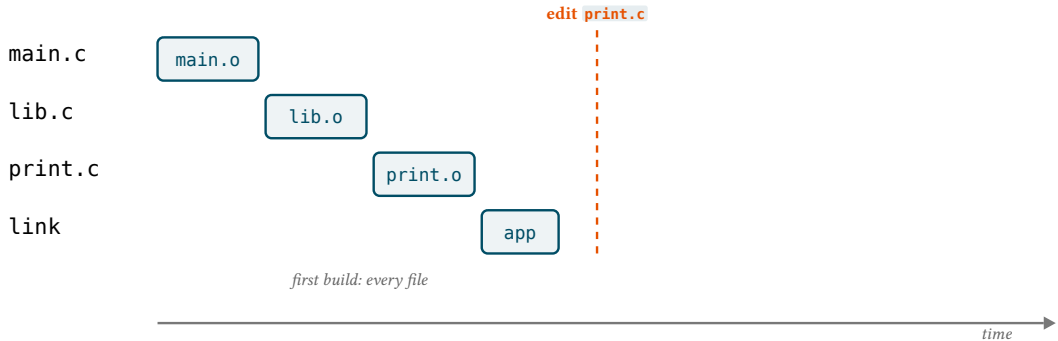



```
$ gcc main.c lib.c print.c -o app
```

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

Compile Everything, Every Time?

`gcc` with a list of source files compiles **all** of them, every time



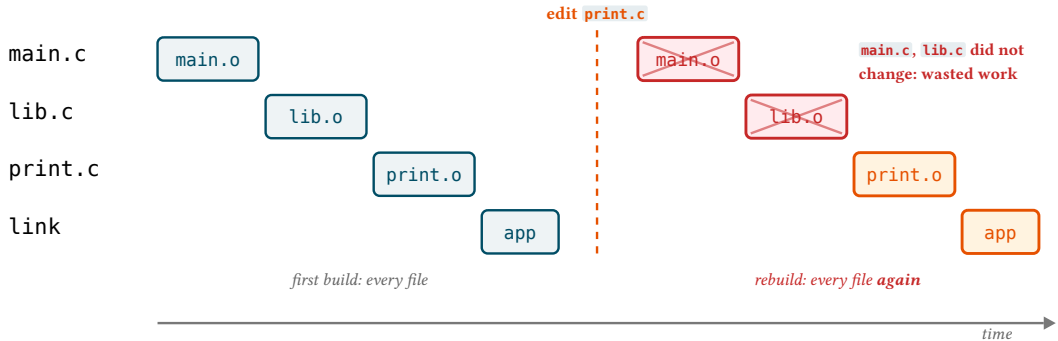

```
$ gcc main.c lib.c print.c -o app
```

```
$ vim print.c                    # edit
```

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

Compile Everything, Every Time?

`gcc` with a list of source files compiles **all** of them, every time




```
$ gcc main.c lib.c print.c -o app
```

```
$ vim print.c                # edit
$ gcc main.c lib.c print.c -o app
```

💡 the Linux kernel has over 30 000 `.c` files: compiling all of them again takes many minutes, for one changed file

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

Several Source Files

Compile each file to an object file, then link them together




```
$ gcc -c main.c
```

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

Several Source Files

Compile each file to an object file, then link them together




```
$ gcc -c main.c
$ gcc -c lib.c
```

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

Several Source Files

Compile each file to an object file, then link them together

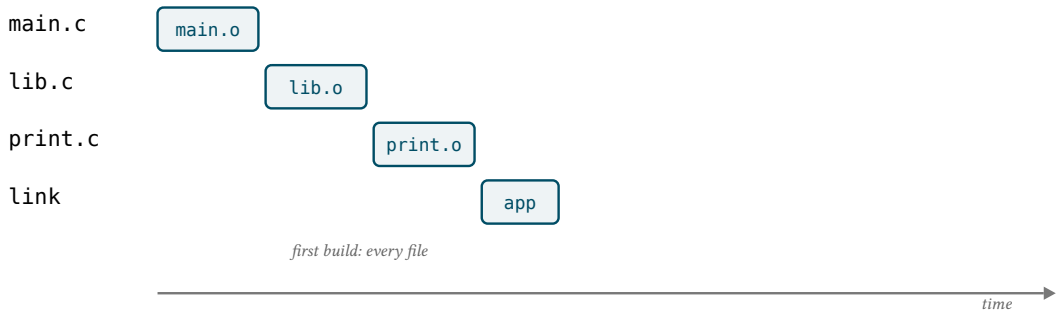



```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
```

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

Several Source Files

Compile each file to an object file, then link them together

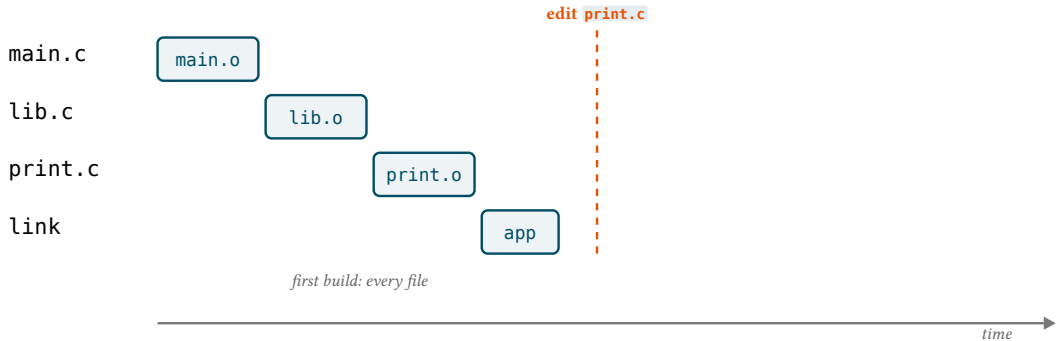



```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

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

Several Source Files

Compile each file to an object file, then link them together



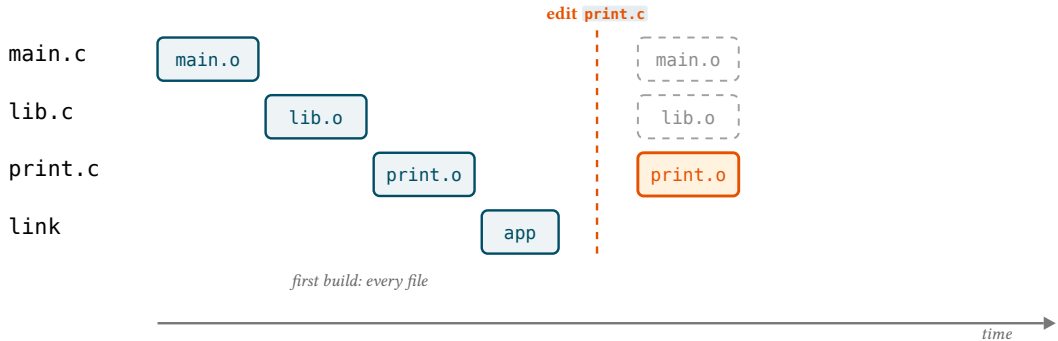

```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

```
$ vim print.c                # edit
```

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

Several Source Files

Compile each file to an object file, then link them together



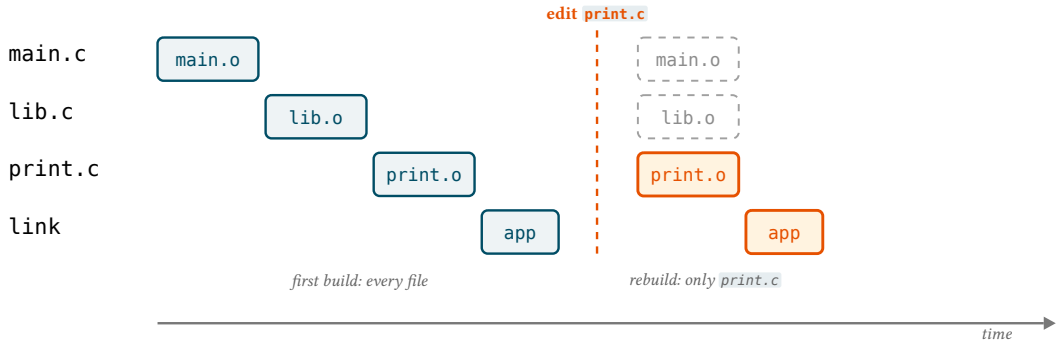

```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

```
$ vim print.c                # edit
$ gcc -c print.c
```

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

Several Source Files

Compile each file to an object file, then link them together




```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

```
$ vim print.c                # edit
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

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

Build Errors

Who reports the error tells you what is wrong

problem	error
missing `.h` preprocessor, `gcc -E` ✅ fix `#include` or `-I dir`	`$ gcc -c main.c` `main.c: fatal error: lib.h: No such file or directory`

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

Build Errors

Who reports the error tells you what is wrong

problem	error
missing `.h` preprocessor, `gcc -E` ✅ fix `#include` or `-I dir`	`$ gcc -c main.c` `main.c: fatal error: lib.h: No such file or directory`
missing prototype compiler, `gcc -S` ✅ declare it in `lib.h`	`$ gcc -c main.c  # GCC 14 and later: an error` `main.c: warning: implicit declaration of function ‘sum’`

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

Build Errors

Who reports the error tells you what is wrong

problem	error
missing `.h` preprocessor, `gcc -E` ✅ fix `#include` or `-I dir`	`$ gcc -c main.c` `main.c: fatal error: lib.h: No such file or directory`
missing prototype compiler, `gcc -S` ✅ declare it in `lib.h`	`$ gcc -c main.c  # GCC 14 and later: an error` `main.c: warning: implicit declaration of function ‘sum’`
missing `.o` linker, `ld` ✅ add `lib.o`	`$ gcc main.o print.o -o app` `main.c:(.text+0x12): undefined reference to `sum'`

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

Build Errors

Who reports the error tells you what is wrong

problem	error
missing `.h` preprocessor, `gcc -E` ✅ fix `#include` or `-I dir`	`$ gcc -c main.c` `main.c: fatal error: lib.h: No such file or directory`
missing prototype compiler, `gcc -S` ✅ declare it in `lib.h`	`$ gcc -c main.c  # GCC 14 and later: an error` `main.c: warning: implicit declaration of function ‘sum’`
missing `.o` linker, `ld` ✅ add `lib.o`	`$ gcc main.o print.o -o app` `main.c:(.text+0x12): undefined reference to `sum'`
missing library linker, `ld` ✅ add `-lm` (`libm`)	`$ gcc root.o -o root` `root.c:(.text+0x18): undefined reference to `sqrt'`

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

Stop After Each Step

command	stops after	writes	look at it with
`gcc -E hello.c -o hello.i`	preprocessor	`hello.i`, C	`less`
`gcc -S hello.c`	compiler	`hello.s`, assembly	`less`
`gcc -c hello.c`	assembler	`hello.o`, object file	`nm`, `objdump -d`
`gcc hello.o -o hello`	linker	`hello`, executable	`file`, `ldd`, `./hello`

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

Stop After Each Step

command	stops after	writes	look at it with
`gcc -E hello.c -o hello.i`	preprocessor	`hello.i`, C	`less`
`gcc -S hello.c`	compiler	`hello.s`, assembly	`less`
`gcc -c hello.c`	assembler	`hello.o`, object file	`nm`, `objdump -d`
`gcc hello.o -o hello`	linker	`hello`, executable	`file`, `ldd`, `./hello`

```
$ gcc -save-temps hello.c -o hello
$ ls
hello  hello.c  hello.i  hello.o  hello.s
```

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

Stop After Each Step

command	stops after	writes	look at it with
`gcc -E hello.c -o hello.i`	preprocessor	`hello.i`, C	`less`
`gcc -S hello.c`	compiler	`hello.s`, assembly	`less`
`gcc -c hello.c`	assembler	`hello.o`, object file	`nm`, `objdump -d`
`gcc hello.o -o hello`	linker	`hello`, executable	`file`, `ldd`, `./hello`

```
$ gcc -save-temps hello.c -o hello
$ ls
hello  hello.c  hello.i  hello.o  hello.s
```

💡 `gcc` is a *driver*: `gcc -v` shows the programs it runs, `cc1`, `as` and `collect2` (which runs `ld`)

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

Building with `make`

Automate the build: run only the commands that are needed

Bibliography

For this section

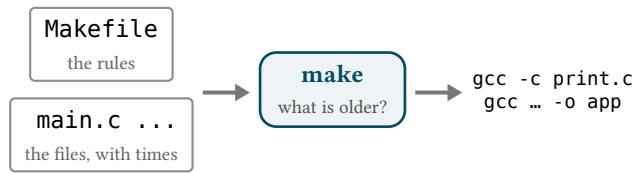
1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.2 - *make*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.6.7 - *Automatizarea procesului de dezvoltare*
     - Section 6.7.1 - *Automatizarea compilării în C/C++*
3. **Richard M. Stallman, Roland McGrath, Paul D. Smith**, *GNU Make Manual*, online
   - Chapter 2 - *An Introduction to Makefiles*
   - Chapter 10 - *Using Implicit Rules* (pattern rules, automatic variables)

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

What Is `make`?

A tool that *automates* the build

- written in 1976 at Bell Labs, by Stuart Feldman; Linux uses **GNU Make**
- reads a `Makefile` from the current directory
- a `Makefile` says, for every file
  - what it is **built from**
  - **which commands** build it
- `make` looks at the **modification time** of the files and runs only the commands that are needed
- not only for C: the slides of this course are built with a `Makefile`



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

A Rule

A `Makefile` is a list of rules

```
target: prerequisites
TAB command 1
TAB command 2
TAB …
```

- **target**: the file to build
- **prerequisites**: the files it is built from
- **recipe**: the commands that build it, one per line, each starting with a TAB

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

A Rule

A `Makefile` is a list of rules

```
target: prerequisites
TAB command 1
TAB command 2
TAB …
```

- **target**: the file to build
- **prerequisites**: the files it is built from
- **recipe**: the commands that build it, one per line, each starting with a TAB

for example: `hello` is built from `hello.c` with **two** commands, run in order

```
hello: hello.c
TAB gcc -c hello.c
TAB gcc hello.o -o hello
```

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

A Rule

A `Makefile` is a list of rules

```
target: prerequisites
TAB command 1
TAB command 2
TAB …
```

- **target**: the file to build
- **prerequisites**: the files it is built from
- **recipe**: the commands that build it, one per line, each starting with a TAB

```
hello: hello.c
TAB gcc -c hello.c
TAB gcc hello.o -o hello
```

for example: `hello` is built from `hello.c` with

⚠ The recipe starts with a TAB, not with spaces

```
hello.o: hello.c
····gcc -c hello.c
```

4 spaces, not a TAB

```
$ make
Makefile:2: *** missing separator.  Stop.
```

💡 set the editor to keep TABs in a `Makefile`

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

A `Makefile` with Two Targets

The `hello.c` program of the previous section

Makefile

```
1 hello: hello.o
2     gcc hello.o -o hello
3
4 hello.o: hello.c
5     gcc -c hello.c
```

- `hello` is **linked** from `hello.o`
- `hello.o` is **compiled** from `hello.c`
- `make` alone builds the **first** target, `hello`

```
hello/
├── Makefile
└── hello.c
```

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

`make`, Step by Step: a New Build

Nothing is built yet

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
```

`hello`

does not exist

🎯 no target given: build the **first** one, `hello`

target stack

▸ `hello` *(goal)*

hello/

```
├── Makefile 11:41:08
└── hello.c  11:41:08
```

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

`make`, Step by Step: a New Build

Nothing is built yet

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
```



`hello` needs `hello.o`, which has its own rule: make it first

target stack

```
▸ hello.o
  hello (goal)
```

hello/

```
├── Makefile  11:41:08
└── hello.c   11:41:08
```

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

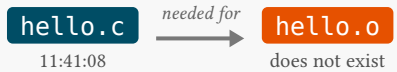
`make`, Step by Step: a New Build

Nothing is built yet

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
```



`hello.c` has no rule: the file must exist

✅ it exists

target stack

- ▸ `hello.c`
- `hello.o`
- `hello` *(goal)*

`hello/`

- `Makefile` 11:41:08
- `hello.c` 11:41:08

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

`make`, Step by Step: a New Build

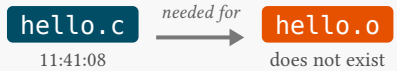


Nothing is built yet

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
```



`hello.o` does not exist: run the recipe

target stack

- ▸ `hello.o`
- `hello` *(goal)*

`hello/`

- `Makefile` 11:41:08
- `hello.c` 11:41:08

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

`make`, Step by Step: a New Build

Nothing is built yet

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
gcc -c hello.c
```



`hello.o` is built

runs

```
gcc -c hello.c
```

target stack

```
▸ hello.o
  hello (goal)
```

hello/

```
├── Makefile  11:41:08
├── hello.c   11:41:08
└── hello.o   11:41:10  built
```

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

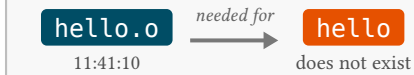
`make`, Step by Step: a New Build

Nothing is built yet

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
gcc -c hello.c
```



back to `hello`: it does not exist, run the recipe

target stack

▸ `hello` *(goal)*

hello/

```
├── Makefile 11:41:08
├── hello.c  11:41:08
└── hello.o  11:41:10 built
```

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

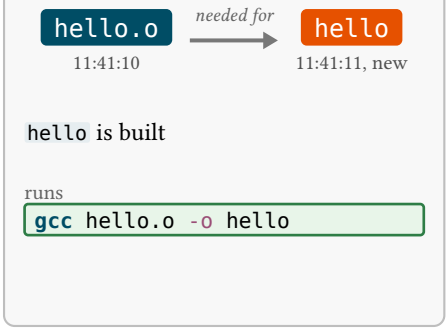
`make`, Step by Step: a New Build

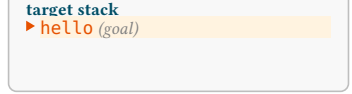
Nothing is built yet

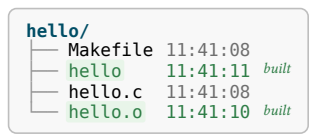
Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
```







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

`make`, Step by Step: Up to Date

Run `make` again, right after the first build

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
$ make
```

`hello`

11:41:11

🎯 no target given: build the **first** one, `hello`

target stack

▸ `hello` *(goal)*

hello/

```
├── Makefile 11:41:08
├── hello    11:41:11
├── hello.c  11:41:08
└── hello.o  11:41:10
```

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

`make`, Step by Step: Up to Date

Run `make` again, right after the first build

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
$ make
```



`hello` needs `hello.o`, which has its own rule: check it first

target stack

▸ `hello.o`
`hello` *(goal)*

hello/

```
├── Makefile 11:41:08
├── hello    11:41:11
├── hello.c  11:41:08
└── hello.o  11:41:10
```

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

`make`, Step by Step: Up to Date

Run `make` again, right after the first build

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
$ make
```



`hello.c` has no rule: the file must exist

✅ it exists

target stack

- ▸ hello.c
- hello.o
- hello *(goal)*

hello/

Makefile	11:41:08
hello	11:41:11
hello.c	11:41:08
hello.o	11:41:10

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

`make`, Step by Step: Up to Date

Run `make` again, right after the first build

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
$ make
```



`hello.c` is **older** than `hello.o`

✅ nothing to run

target stack

- ▸ `hello.o`
- `hello` *(goal)*

`hello/`

`Makefile`	`11:41:08`
`hello`	`11:41:11`
`hello.c`	`11:41:08`
`hello.o`	`11:41:10`

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

`make`, Step by Step: Up to Date

Run `make` again, right after the first build

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ make
gcc -c hello.c
gcc hello.o -o hello
$ make
make: 'hello' is up to date.
```



back to `hello`: `hello.o` is **older** than `hello`

✅ nothing to run, `hello` is up to date

target stack

▸ `hello` *(goal)*

hello/

```
├── Makefile 11:41:08
├── hello    11:41:11
├── hello.c  11:41:08
└── hello.o  11:41:10
```

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ touch hello.c   # as if edited
```

`hello.c`

11:41:13, now

`touch` sets the time of `hello.c` to **now**, as if we edited it

target stack

empty: `make` is not running

hello/

```
├── Makefile 11:41:08
├── hello    11:41:11
├── hello.c  11:41:13  edited
└── hello.o  11:41:10
```

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ touch hello.c   # as if edited
$ make
```



🎯 `hello` needs `hello.o`, which has its own rule: make it first

target stack

```
▸ hello.o
  hello (goal)
```

hello/

```
├── Makefile 11:41:08
├── hello    11:41:11
├── hello.c  11:41:13  edited
└── hello.o  11:41:10
```

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ touch hello.c   # as if edited
$ make
```



`hello.c` has no rule: the file must exist

✅ it exists

target stack

- ▸ `hello.c`
- `hello.o`
- `hello` *(goal)*

`hello/`

`Makefile`	`11:41:08`	
`hello`	`11:41:11`	
`hello.c`	`11:41:13`	*edited*
`hello.o`	`11:41:10`	

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

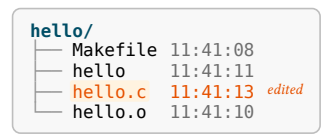
```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ touch hello.c   # as if edited
$ make
```



`hello.c` is **newer** than `hello.o`: run the recipe





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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ touch hello.c    # as if edited
$ make
gcc -c hello.c
```



`hello.o` is built again

runs

```
gcc -c hello.c
```

target stack

- ▶ `hello.o`
- `hello` *(goal)*

hello/

Makefile	11:41:08	
hello	11:41:11	
hello.c	11:41:13	*edited*
hello.o	11:41:13	*built*

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1 ▸hello: hello.o
2       gcc hello.o -o hello
3
4  hello.o: hello.c
5       gcc -c hello.c
```

```
$ touch hello.c   # as if edited
$ make
gcc -c hello.c
```



back to `hello`: `hello.o` is now **newer**, run the recipe

target stack

▸ `hello` *(goal)*

hello/

Makefile	11:41:08	
hello	11:41:11	
hello.c	11:41:13	*edited*
hello.o	11:41:13	*built*

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

`make`, Step by Step: a Modified File

Change the source file and build again

Makefile

```
1  hello: hello.o
2      gcc hello.o -o hello
3
4  hello.o: hello.c
5      gcc -c hello.c
```

```
$ touch hello.c   # as if edited
$ make
gcc -c hello.c
gcc hello.o -o hello
```



`hello` is built again

runs

```
gcc hello.o -o hello
```

target stack

▶ `hello` *(goal)*

hello/

Makefile	11:41:08	
hello	11:41:13	*built*
hello.c	11:41:13	*edited*
hello.o	11:41:13	*built*

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

How `make` Decides

A summary of the steps: `make --debug=v` prints them

to make a target

1. **push** it on the target stack
2. make each **prerequisite** first, in the same way

```
$ touch hello.c
$ make --debug=v
...
Considering target file 'hello'.
  Considering target file 'hello.o'.
    Considering target file 'hello.c'.
     Finished prerequisites of target file 'hello.c'.
    No need to remake target 'hello.c'.
   Finished prerequisites of target file 'hello.o'.
   Prerequisite 'hello.c' is newer than target 'hello.o'.
  Must remake target 'hello.o'.
gcc -c hello.c
  Successfully remade target file 'hello.o'.
 Finished prerequisites of target file 'hello'.
 Prerequisite 'hello.o' is newer than target 'hello'.
Must remake target 'hello'.
gcc hello.o -o hello
Successfully remade target file 'hello'.
```

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

How `make` Decides

A summary of the steps: `make --debug=v` prints them

to make a target

1. **push** it on the target stack
2. make each **prerequisite** first, in the same way
3. missing, or a prerequisite is **newer**? → run the recipe

```
$ touch hello.c
$ make --debug=v
...
Considering target file 'hello'.
  Considering target file 'hello.o'.
    Considering target file 'hello.c'.
     Finished prerequisites of target file 'hello.c'.
    No need to remake target 'hello.c'.
   Finished prerequisites of target file 'hello.o'.
   Prerequisite 'hello.c' is newer than target 'hello.o'.
  Must remake target 'hello.o'.
gcc -c hello.c
  Successfully remade target file 'hello.o'.
 Finished prerequisites of target file 'hello'.
 Prerequisite 'hello.o' is newer than target 'hello'.
Must remake target 'hello'.
gcc hello.o -o hello
Successfully remade target file 'hello'.
```

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

How `make` Decides

A summary of the steps: `make --debug=v` prints them

to make a target

1. **push** it on the target stack
2. make each **prerequisite** first, in the same way
3. missing, or a prerequisite is **newer**? → run the recipe
4. otherwise it is *up to date*: nothing to run

```
$ touch hello.c
$ make --debug=v
...
Considering target file 'hello'.
  Considering target file 'hello.o'.
    Considering target file 'hello.c'.
     Finished prerequisites of target file 'hello.c'.
    No need to remake target 'hello.c'.
   Finished prerequisites of target file 'hello.o'.
   Prerequisite 'hello.c' is newer than target 'hello.o'.
  Must remake target 'hello.o'.
gcc -c hello.c
  Successfully remade target file 'hello.o'.
 Finished prerequisites of target file 'hello'.
 Prerequisite 'hello.o' is newer than target 'hello'.
Must remake target 'hello'.
gcc hello.o -o hello
Successfully remade target file 'hello'.
```

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

How `make` Decides

A summary of the steps: `make --debug=v` prints them

to make a target

1. **push** it on the target stack
2. make each **prerequisite** first, in the same way
3. missing, or a prerequisite is **newer**? → run the recipe
4. otherwise it is *up to date*: nothing to run
5. **pop** it, back to the target below

💡 `make` reads only the **times** of the files, never the code

```
$ touch hello.c
$ make --debug=v
...
Considering target file 'hello'.
  Considering target file 'hello.o'.
    Considering target file 'hello.c'.
     Finished prerequisites of target file 'hello.c'.
    No need to remake target 'hello.c'.
   Finished prerequisites of target file 'hello.o'.
   Prerequisite 'hello.c' is newer than target 'hello.o'.
  Must remake target 'hello.o'.
gcc -c hello.c
  Successfully remade target file 'hello.o'.
 Finished prerequisites of target file 'hello'.
 Prerequisite 'hello.o' is newer than target 'hello'.
Must remake target 'hello'.
gcc hello.o -o hello
Successfully remade target file 'hello'.
```

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

How `make` Decides

A summary of the steps: `make --debug=v` prints them

to make a target

1. **push** it on the target stack
2. make each **prerequisite** first, in the same way
3. missing, or a prerequisite is **newer**? → run the recipe
4. otherwise it is *up to date*: nothing to run
5. **pop** it, back to the target below

💡 `make` reads only the **times** of the files, never the code

```
$ touch hello.c
$ make --debug=v
...
Considering target file 'hello'.
  Considering target file 'hello.o'.
    Considering target file 'hello.c'.
     Finished prerequisites of target file 'hello.c'.
    No need to remake target 'hello.c'.
   Finished prerequisites of target file 'hello.o'.
   Prerequisite 'hello.c' is newer than target 'hello.o'.
  Must remake target 'hello.o'.
gcc -c hello.c
  Successfully remade target file 'hello.o'.
 Finished prerequisites of target file 'hello'.
 Prerequisite 'hello.o' is newer than target 'hello'.
Must remake target 'hello'.
gcc hello.o -o hello
Successfully remade target file 'hello'.
```

the indentation is the **target stack**: one level for every target `make` is working on

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

A `Makefile` for a Project

The same steps for several files, a shorter `Makefile`

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.2 - *make*
2. **Razvan Deaconescu, Razvan Rughinis, Mihai Carabas, Alexandru Radovici**, *Utilizarea Sistemelor de Operare*, Printech 2021, download
   - Chapter 6 - *Dezvoltarea aplicațiilor*
     - Section 6.7.1 - *Automatizarea compilării în C/C++*
3. **Richard M. Stallman, Roland McGrath, Paul D. Smith**, *GNU Make Manual*, online
   - Chapter 2 - *An Introduction to Makefiles*
   - Chapter 10 - *Using Implicit Rules* (pattern rules, automatic variables)

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

The Project

The program of *Several Source Files*: `main.c` uses two *modules*, `lib` and `print`

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

💡 a module is a pair of files: the `.h` **declares** what it offers, the `.c` **defines** how it works

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

The Project

The program of *Several Source Files*: `main.c` uses two *modules*, `lib` and `print`

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

💡 `main.c` calls `sum`: declared in `lib.h`, defined in `lib.c`, adds the numbers from 1 to `MAX`

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

The Project

The program of *Several Source Files*: `main.c` uses two *modules*, `lib` and `print`

main.c

```
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  }
```

lib.h

```
1  #ifndef LIB_H
2  #define LIB_H
3  #define MAX 10
4  int sum(int n);
5  #endif
```

print.h

```
1  #ifndef PRINT_H
2  #define PRINT_H
3  void print_result(const char *label,
4                    int value);
5  #endif
```

lib.c

```
1  #include "lib.h"
2
3  int sum(int n)
4  {
5      int s = 0;
6      for (int i = 1; i <= n; i++) {
7          s = s + i;
8      }
9      return s;
10 }
```

print.c

```
1  #include <stdio.h>
2  #include "print.h"
3
4  void print_result(const char *label,
5                    int value)
6  {
7      printf("%s = %d\n", label, value);
8  }
```

💡 `main.c` calls `print_result`: declared in `print.h`, defined in `print.c`, uses `printf`

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

The Project

The program of *Several Source Files*: `main.c` uses two *modules*, `lib` and `print`

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
```

💡 each `.c` is **compiled** to its own `.o`, then the three `.o` files are **linked** into `app`

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

The Project

The program of *Several Source Files*: `main.c` uses two *modules*, `lib` and `print`

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

```
$ gcc -c main.c
$ gcc -c lib.c
$ gcc -c print.c
$ gcc main.o lib.o print.o -o app
$ ./app
sum = 55
```

💡 four commands to remember and type again after every change: `make` will run them for us

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

A `Makefile` for the Project

One rule for every file we build, the `.h` files come in a later section

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

- `app` is **linked** from the three object files
- every object file is **compiled** from its `.c`
- `make` alone builds the **first** target, `app`

```
project/
├── Makefile
├── lib.c
├── lib.h
├── main.c
├── print.c
└── print.h
```

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

The Project, Step by Step: a New Build

The same steps as for `hello`, for every object file

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
```

app

does not exist

🎯 no target given: build the **first** one, `app`

target stack

▶ app *(goal)*

project/

```
project/
├── Makefile
├── lib.c
├── lib.h
├── main.c
├── print.c
└── print.h
```

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

The Project, Step by Step: a New Build



The same steps as for `hello`, for every object file

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
```

`main.o` *needed for* `app`

`main.o` does not exist — `app` does not exist

`app` needs `main.o`, `lib.o` and `print.o`: make them first, one by one

target stack

▶ app *(goal)*

project/

- Makefile
- lib.c
- lib.h
- main.c
- print.c
- print.h

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

The Project, Step by Step: a New Build

The same steps as for `hello`, for every object file

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
gcc -c main.c
```

`main.c` 12:09:11 — *needed for* → `main.o` 12:09:13, new

`main.c` exists ✅, `main.o` does not exist

runs

```
gcc -c main.c
```

target stack

- ▶ `main.o`
- `app` *(goal)*

project/

- Makefile
- lib.c
- lib.h
- main.c
- main.o *built*
- print.c
- print.h

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

The Project, Step by Step: a New Build

The same steps as for `hello`, for every object file

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
gcc -c main.c
gcc -c lib.c
```



`lib.c` exists ✅, `lib.o` does not exist

runs

```
gcc -c lib.c
```

target stack

- ▸ `lib.o`
- `app` *(goal)*

project/

- Makefile
- lib.c
- lib.h
- `lib.o` *built*
- main.c
- `main.o` *built*
- print.c
- print.h

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

The Project, Step by Step: a New Build



The same steps as for `hello`, for every object file

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
gcc -c main.c
gcc -c lib.c
gcc -c print.c
```



`print.c` exists ✅, `print.o` does not exist

runs

```
gcc -c print.c
```

target stack

- ▸ print.o
- app *(goal)*

project/

- Makefile
- lib.c
- lib.h
- lib.o *built*
- main.c
- main.o *built*
- print.c
- print.h
- print.o *built*

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

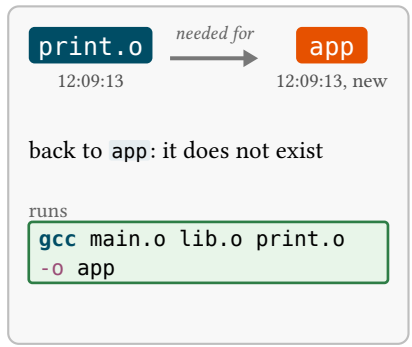
The Project, Step by Step: a New Build

The same steps as for `hello`, for every object file

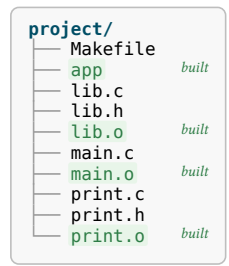
Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ make
gcc -c main.c
gcc -c lib.c
gcc -c print.c
gcc main.o lib.o print.o -o app
```







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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
```

print.c

12:09:15, now

touch sets the time of `print.c` to **now**, as if we edited it

target stack

empty: ***make*** *is not running*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c *edited*
- print.h
- print.o

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

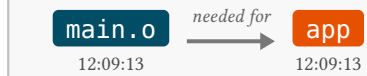
The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
$ make
```



🎯 `app` needs `main.o`, `lib.o` and `print.o`: check them first

target stack

▶ app *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c *edited*
- print.h
- print.o

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
$ make
```



`main.c` is **older** than `main.o`

✅ nothing to run

target stack

- ▶ `main.o`
- `app` *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c *edited*
- print.h
- print.o

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
$ make
```



`lib.c` is **older** than `lib.o`

✅ nothing to run

target stack

- ▸ `lib.o`
- `app` *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c *edited*
- print.h
- print.o

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
$ make
gcc -c print.c
```



`print.c` is **newer** than `print.o`

runs

```
gcc -c print.c
```

target stack

- `print.o`
- `app` *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- `print.c` *edited*
- print.h
- `print.o` *built*

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

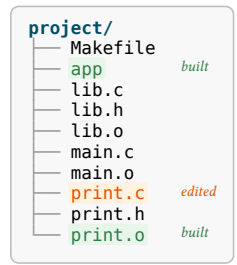
Makefile

```
1  app: main.o lib.o print.o
2      gcc main.o lib.o print.o -o app
3
4  main.o: main.c
5      gcc -c main.c
6
7  lib.o: lib.c
8      gcc -c lib.c
9
10 print.o: print.c
11     gcc -c print.c
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```







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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc $^ -o $@
3
4  main.o: main.c
5      gcc -c $< -o $@
6
7  lib.o: lib.c
8      gcc -c $< -o $@
9
10 print.o: print.c
11     gcc -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

variable	means
`$@`	the **target** of the rule
`$<`	the **first** prerequisite
`$^`	**all** the prerequisites

💡 `make` fills them in, for each rule, **inside the recipe**

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app                 built
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c             edited
├── print.h
└── print.o             built
```

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc $^ -o $@
3
4  main.o: main.c
5      gcc -c $< -o $@
6
7  lib.o: lib.c
8      gcc -c $< -o $@
9
10 print.o: print.c
11     gcc -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

we write

```
gcc $^ -o $@
```

`make` runs

```
gcc main.o lib.o print.o
-o app
```

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app            built
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c        edited
├── print.h
└── print.o        built
```

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc $^ -o $@
3
4  main.o: main.c
5      gcc -c $< -o $@
6
7  lib.o: lib.c
8      gcc -c $< -o $@
9
10 print.o: print.c
11     gcc -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

we write

```
gcc -c $< -o $@
```

`make` runs

```
gcc -c main.c -o main.o
```

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app                built
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c            edited
├── print.h
└── print.o            built
```

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc $^ -o $@
3
4  main.o: main.c
5      gcc -c $< -o $@
6
7  lib.o: lib.c
8      gcc -c $< -o $@
9
10 print.o: print.c
11     gcc -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

the three recipes of the object files are now the **same**:

```
gcc -c $< -o $@
```

target stack

empty: `make` *is not running*

```
project/
├── Makefile
├── app          built
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c      edited
├── print.h
└── print.o      built
```

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  app: main.o lib.o print.o
2      gcc $^ -o $@
3
4  %.o: %.c
5      gcc -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

a **pattern rule**: `%` matches any name, so `%.o: %.c` means

`main.o: main.c`

`lib.o: lib.c`

`print.o: print.c`

✅ 11 lines → 5 lines

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app              built
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c          edited
├── print.h
└── print.o          built
```

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

The Project, Step by Step: a Modified File

Rebuild only what changed, then shorten the `Makefile`: automatic variables, a pattern rule, variables

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ touch print.c
$ make
gcc -c print.c
gcc main.o lib.o print.o -o app
```

we write

`$(CC)` `$(CFLAGS)` `-c $< -o $@`

`make` runs

`gcc` `-Wall -g` `-c main.c -o main.o`

target stack

empty: `make` is not running

project/

- Makefile
- app — *built*
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c — *edited*
- print.h
- print.o — *built*

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

Special Targets

Targets that are not files: `clean`, `all`, `build`, `run`

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.2 - *make*
2. **Richard M. Stallman, Roland McGrath, Paul D. Smith**, *GNU Make Manual*, online
   - Chapter 4 - *Writing Rules*
     - *Phony Targets*
     - *Special Built-in Target Names*
   - Chapter 16 - *Makefile Conventions*
     - *Standard Targets for Users*

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

`clean`: a Target That Is Not a File

It has no prerequisites and its recipe builds no file

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ make clean
```

`clean`

does not exist

🎯 the goal is `clean`, it has **no** prerequisites

target stack

▸ `clean` *(goal)*

project/

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

`clean`: a Target That Is Not a File

It has no prerequisites and its recipe builds no file

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ make clean
rm -f *.o app
```

`clean`

does not exist

there is no file named `clean`: run the recipe

runs

```
rm -f *.o app
```

target stack

▸ `clean` *(goal)*

project/

- Makefile
- app — *deleted*
- lib.c
- lib.h
- lib.o — *deleted*
- main.c
- main.o — *deleted*
- print.c
- print.h
- print.o — *deleted*

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

`clean`: a Target That Is Not a File

It has no prerequisites and its recipe builds no file

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ make clean
rm -f *.o app
$ make clean
rm -f *.o app
```

`clean`

does not exist

again: still no file named `clean`, the recipe runs **every time**

runs

```
rm -f *.o app
```

target stack

▸ `clean` *(goal)*

```
project/
├── Makefile
├── lib.c
├── lib.h
├── main.c
├── print.c
└── print.h
```

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

A File Named `clean`

`make` cannot tell a target from a file with the same name

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ touch clean
```

`clean`

file, in `project/`

a file named `clean` appears in the folder: created by mistake, by a script, ...

target stack

empty: `make` is not running

project/

```
project/
├── Makefile
├── app
├── clean        new file
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

A File Named `clean`

`make` cannot tell a target from a file with the same name

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ touch clean
```

`clean` target, in `Makefile` = `clean` file, in `project/`

the **same name**: for `make`, a target is the **file** it builds, so the target `clean` now **is** this file

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app
├── clean        new file
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

A File Named `clean`

`make` cannot tell a target from a file with the same name

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ touch clean
$ make clean
make: 'clean' is up to date.
```

clean

the file exists

the file `clean` exists and has no prerequisites: it is *up to date*

❌ the recipe does not run

target stack

▸ clean *(goal)*

```
project/
├── Makefile
├── app
├── clean          new file
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

A File Named `clean`

`make` cannot tell a target from a file with the same name

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 clean:
11     rm -f *.o app
```

```
$ touch clean
$ make clean
make: 'clean' is up to date.
```

`clean`

the file exists

the file `clean` exists and has no prerequisites: it is *up to date*

❌ the recipe does not run

target stack

▸ `clean` *(goal)*

project/

- Makefile
- app
- clean *new file*
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

⚠ Nothing was deleted

`make clean` did not run `rm: app` and the `.o` files are **still here**

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

.PHONY

Tell `make` which targets are not files

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 .PHONY: clean
11 clean:
12     rm -f *.o app
```

clean

phony

`.PHONY` is a **special target**: its prerequisites are *phony* targets, not files

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app
├── clean
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

.PHONY

Tell `make` which targets are not files

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 .PHONY: clean
11 clean:
12     rm -f *.o app
```

```
$ ls clean
clean
$ make clean
rm -f *.o app
```

clean

phony

`clean` is phony: no file is checked, the recipe **always** runs

runs

```
rm -f *.o app
```

target stack

▸ clean *(goal)*

project/

- Makefile
- app *deleted*
- clean
- lib.c
- lib.h
- lib.o *deleted*
- main.c
- main.o *deleted*
- print.c
- print.h
- print.o *deleted*

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

`all`, `build` and `run`

Phony targets give names to what we ask `make` to do

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

`all` `build` `run` `clean`

one `.PHONY` line for **all** the targets that are not files: their recipes always run

we type	`make`

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

`all`, `build` and `run`

Phony targets give names to what we ask `make` to do

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```



`all` is the **first** target, so `make` means `make all`; by convention, it builds **everything**

we type	`make`
`make`	builds `app`

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

`all`, `build` and `run`

Phony targets give names to what we ask `make` to do

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```



`build` builds the program, `app` it has no recipe of its own

we type	`make`
`make`	builds `app`
`make build`	builds `app`

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

`all`, `build` and `run`

Phony targets give names to what we ask `make` to do

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```



`run` builds the program first, then starts it

runs

```
./app
```

we type	`make`
`make`	builds `app`
`make build`	builds `app`
`make run`	builds and runs `app`

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

`all`, `build` and `run`

Phony targets give names to what we ask `make` to do

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

`clean`

phony

`clean` deletes what `make` built

runs

```
rm -f *.o app
```

we type	`make`
`make`	builds `app`
`make build`	builds `app`
`make run`	builds and runs `app`
`make clean`	deletes what it built

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

`make run`, Step by Step

`app` is already built

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

```
$ make run
```



🎯 the goal is `run`: it needs `build`, make it first

target stack

▶ `run` *(goal)*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

`make run`, Step by Step

`app` is already built

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

```
$ make run
```



`build` needs `app`: make it first

target stack
- ▸ build
- run *(goal)*

project/
- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

`make run`, Step by Step

`app` is already built

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

```
$ make run
```



the same checks as before: the object files and `app` are up to date ✅

target stack

- ▸ app
- build
- run *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

`make run`, Step by Step

`app` is already built

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

```
$ make run
```



back to `build`: it has no recipe, nothing to run

target stack

- ▸ build
- run *(goal)*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

`make run`, Step by Step

`app` is already built

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  .PHONY: all build run clean
5
6  all: build
7
8  build: app
9
10 run: build
11     ./app
12
13 app: main.o lib.o print.o
14     $(CC) $^ -o $@
15
16 %.o: %.c
17     $(CC) $(CFLAGS) -c $< -o $@
18
19 clean:
20     rm -f *.o app
```

```
$ make run
./app
sum = 55
```



back to `run`: phony, so its recipe always runs

runs

```
./app
```

target stack

▶ run *(goal)*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

Common Phony Targets

The names other programmers expect

target	what it does
`all`	build everything; the **first** target, the default
`build`	build the program (used by many projects)
`run`	build and run the program
`clean`	delete the files `make` built
`test`, `check`	build and run the tests
`install`	copy the program into the system, e.g. `/usr/local/bin`

- `all`, `clean`, `install`, `check` are **GNU conventions**
- write them in `.PHONY`
- a phony target is always *out of date*: its recipe always runs

💡 `make clean && make` builds everything from scratch, like `make -B`

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

Header Dependencies

Where do the `.h` files go in a `Makefile`?

Bibliography

For this section

1. **Brian Ward**, *How LINUX Works*, 3rd Edition, No Starch Press, 2021
   - Chapter 15 - *Development Tools*
     - Section 15.2 - *make*
2. **Richard M. Stallman, Roland McGrath, Paul D. Smith**, *GNU Make Manual*, online
   - Chapter 4 - *Writing Rules*
     - *Generating Prerequisites Automatically*
3. *GCC Manual*, *Options Controlling the Preprocessor* (`-MM`, `-MMD`)

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

Who Includes Whom?

The dependencies come from the `#include` lines of every `.c` file

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```



two `#include` lines: two headers

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

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

Who Includes Whom?

The dependencies come from the `#include` lines of every `.c` file

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```



two `#include` lines: two headers

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```



`lib.c` includes its **own** header: good practice

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

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

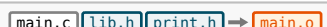
Who Includes Whom?

The dependencies come from the `#include` lines of every `.c` file

main.c

```c
#include "lib.h"
#include "print.h"

int main(void)
{
    int total = sum(MAX);
    print_result("sum", total);
    return 0;
}
```



two `#include` lines: two headers

lib.h

```c
#ifndef LIB_H
#define LIB_H
#define MAX 10
int sum(int n);
#endif
```

lib.c

```c
#include "lib.h"

int sum(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}
```



`lib.c` includes its **own** header: good practice

print.h

```c
#ifndef PRINT_H
#define PRINT_H
void print_result(const char *label,
                  int value);
#endif
```

print.c

```c
#include <stdio.h>
#include "print.h"

void print_result(const char *label,
                  int value)
{
    printf("%s = %d\n", label, value);
}
```

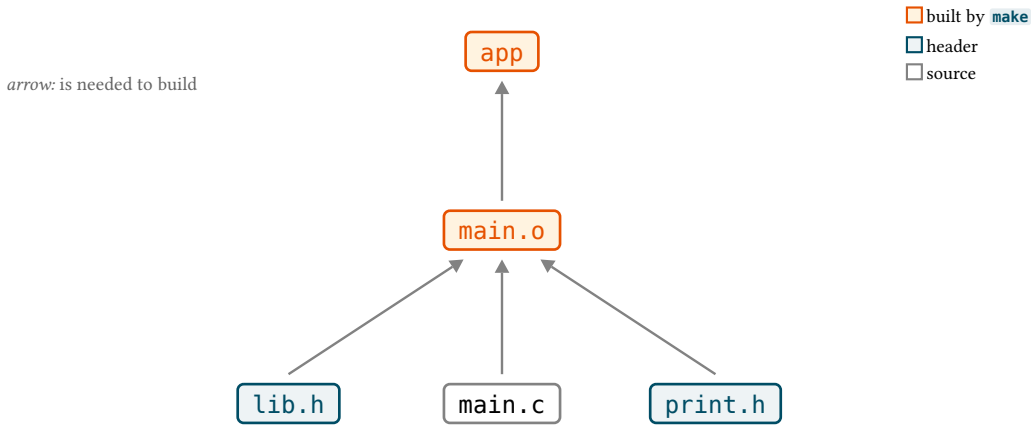


`<stdio.h>` is a system header, it does not change

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

The Dependency Tree

What `make` must know to rebuild the right files

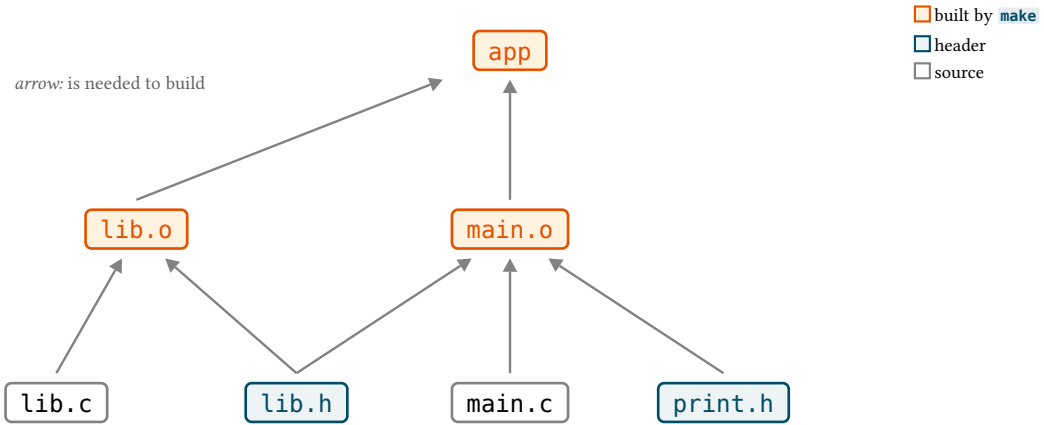


💡 `main.o` is built from `main.c` and from the two headers it includes

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

The Dependency Tree

What `make` must know to rebuild the right files

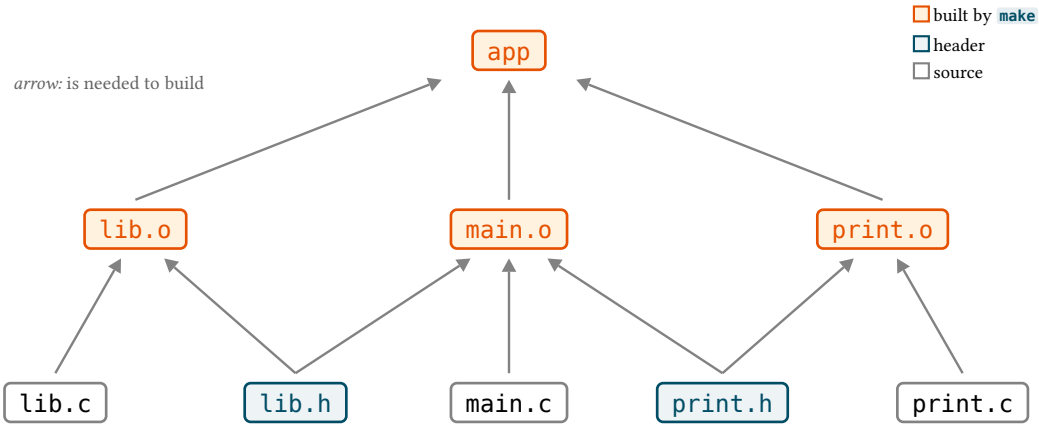


💡 `lib.o` is built from `lib.c` and `lib.h`: `lib.h` is needed by **two** object files

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

The Dependency Tree

What `make` must know to rebuild the right files

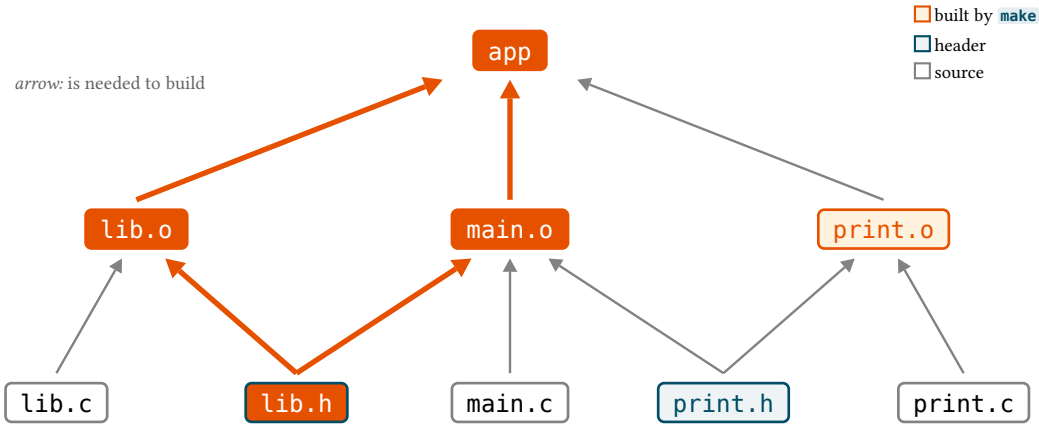


💡 `print.h` is also needed by two object files: `main.o` and `print.o`

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

The Dependency Tree

What `make` must know to rebuild the right files



change `lib.h` → compile `main.o` and `lib.o` again → link `app` again; `print.o` is kept

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

The Bug



Change a header, with the `Makefile` of the previous sections

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

`lib.h`

12:35:59, edited

we change `lib.h`:

```
#define MAX 100
```

target stack

empty: ***`make`*** *is not running*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h        edited
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

The Bug

Change a header, with the `Makefile` of the previous sections

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
```



🎯 `app` needs `main.o`, `lib.o` and `print.o`: check them first

target stack

▸ app *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Bug

Change a header, with the `Makefile` of the previous sections

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
```



`main.c` is **older** than `main.o` ✅

`lib.h` is not in the rule: `make` does not look at it

target stack

- ▸ `main.o`
- `app` *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Bug

Change a header, with the `Makefile` of the previous sections

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
make: 'app' is up to date.
```



back to `app`: all the object files are older

✅ `app` is up to date, nothing to run

target stack

▸ app *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Bug

Change a header, with the `Makefile` of the previous sections

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
make: 'app' is up to date.
$ ./app
sum = 55
```



back to `app`: all the object files are older

✅ `app` is up to date, nothing to run

target stack

empty: ***make*** *is not running*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h        edited
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

⚠ Wrong result: 1 + 2 + ... + 100 is 5050

`main.o` depends on `lib.h` (`main.c` includes it), but the `Makefile` does not say so: `make` does not compile `main.o` again, it still has `MAX = 10`

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

The Wrong Place

Add the header to the rule of `app`?

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o lib.h
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

`lib.h` *needed for* `app`

12:35:59 → 12:35:57

we add `lib.h` to the prerequisites of `app`

target stack

empty: `make` *is not running*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h        edited
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

The Wrong Place

Add the header to the rule of `app`?

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o lib.h
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
```



`main.c` is **older** than `main.o` ✅
nothing to run

target stack
▸ main.o
app *(goal)*

project/
- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Wrong Place

Add the header to the rule of `app`?

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o lib.h
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
gcc main.o lib.o print.o lib.h -o app
```



back to `app`: `lib.h` is **newer**, link again

runs

```
gcc main.o lib.o print.o
lib.h -o app
```

target stack

▸ app *(goal)*

project/

- Makefile
- app — *built*
- lib.c
- lib.h — *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Wrong Place

Add the header to the rule of `app`?

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o lib.h
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
gcc main.o lib.o print.o lib.h -o app
$ ./app
sum = 55
```



back to `app`: `lib.h` is **newer**, link again

runs

```
gcc main.o lib.o print.o
lib.h -o app
```

target stack

empty: `make` *is not running*

```
project/
├── Makefile
├── app          built
├── lib.c
├── lib.h        edited
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

⚠ Still 55: only app was linked again

`app` needs `lib.h` only **through** `main.o` and `lib.o`: the header goes on **their** rules. `main.o` was not compiled again, it still has `MAX = 10`

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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

`lib.h` `print.h`

three rules **without a recipe**: they only **add** prerequisites, the recipe still comes from `%.o: %.c`

target stack

empty: **`make`** *is not running*

```
project/
├── Makefile
├── app
├── lib.c
├── lib.h        edited
├── lib.o
├── main.c
├── main.o
├── print.c
├── print.h
└── print.o
```

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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
```



🎯 `app` needs `main.o`, `lib.o` and `print.o`: check them first

target stack

▸ app *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o
- main.c
- main.o
- print.c
- print.h
- print.o

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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

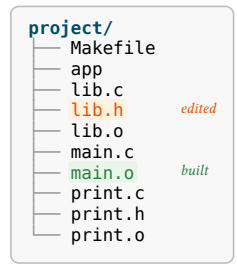
Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
gcc -Wall -g -c main.c -o main.o
```







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

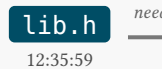
The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c lib.c -o lib.o
```



`lib.h` is **newer** than `lib.o`

runs

```
gcc -Wall -g -c lib.c -o lib.o
```

target stack

- ▸ `lib.o`
- `app` *(goal)*

project/

- Makefile
- app
- lib.c
- lib.h — *edited*
- lib.o — *built*
- main.c
- main.o — *built*
- print.c
- print.h
- print.o

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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c lib.c -o lib.o
```



`print.c` and `print.h` are **older** than `print.o` ✅
nothing to run

target stack
▸ print.o
app *(goal)*

project/
- Makefile
- app
- lib.c
- lib.h *edited*
- lib.o *built*
- main.c
- main.o *built*
- print.c
- print.h
- print.o

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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

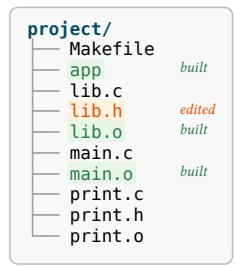
Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c lib.c -o lib.o
gcc main.o lib.o print.o -o app
```







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

The Right Place

Headers go on the rules of the **object files** whose `.c` includes them

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g
3
4  app: main.o lib.o print.o
5      $(CC) $^ -o $@
6
7  %.o: %.c
8      $(CC) $(CFLAGS) -c $< -o $@
9
10 main.o: lib.h print.h
11 lib.o: lib.h
12 print.o: print.h
```

```
$ make
gcc -Wall -g -c main.c -o main.o
gcc -Wall -g -c lib.c -o lib.o
gcc main.o lib.o print.o -o app
$ ./app
sum = 5050
```

app

sum = 5050

✅ `main.o` was compiled with `MAX = 100`, the result is right

💡 **Tip**: a dependency follows the `#include`, not the file name: `lib.o` depends on `lib.h` **because** `lib.c` includes it

target stack

empty: `make` is not running

```
project/
├── Makefile
├── app          built
├── lib.c
├── lib.h        edited
├── lib.o        built
├── main.c
├── main.o       built
├── print.c
├── print.h
└── print.o
```

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

Bonus: Automatic Dependencies

Let `gcc` write the header dependencies

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g -MMD
3  OBJS = main.o lib.o print.o
4
5  app: $(OBJS)
6      $(CC) $^ -o $@
7
8  %.o: %.c
9      $(CC) $(CFLAGS) -c $< -o $@
10
11 -include $(OBJS:.o=.d)
12
13 .PHONY: clean
14 clean:
15     rm -f *.o *.d app
```



`-MMD`: while compiling `X.c`, `gcc` also writes `X.d`, with the headers it read

```
project/
├── Makefile
├── app
├── lib.c
├── lib.d          built
├── lib.h
├── lib.o
├── main.c
├── main.d         built
├── main.o
├── print.c
├── print.d        built
├── print.h
└── print.o
```

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

Bonus: Automatic Dependencies

Let `gcc` write the header dependencies

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g -MMD
3  OBJS = main.o lib.o print.o
4
5  app: $(OBJS)
6      $(CC) $^ -o $@
7
8  %.o: %.c
9      $(CC) $(CFLAGS) -c $< -o $@
10
11 -include $(OBJS:.o=.d)
12
13 .PHONY: clean
14 clean:
15     rm -f *.o *.d app
```

`main.d lib.d print.d`

`$(OBJS:.o=.d)` is

`main.d lib.d print.d`

`-include` reads them, if they exist

(the `-` ignores the missing ones)

```
project/
├── Makefile
├── app
├── lib.c
├── lib.d          built
├── lib.h
├── lib.o
├── main.c
├── main.d         built
├── main.o
├── print.c
├── print.d        built
├── print.h
└── print.o
```

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

Bonus: Automatic Dependencies

Let `gcc` write the header dependencies

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -g -MMD
3  OBJS = main.o lib.o print.o
4
5  app: $(OBJS)
6      $(CC) $^ -o $@
7
8  %.o: %.c
9      $(CC) $(CFLAGS) -c $< -o $@
10
11 -include $(OBJS:.o=.d)
12
13 .PHONY: clean
14 clean:
15     rm -f *.o *.d app
```

```
$ cat main.d lib.d print.d
main.o: main.c lib.h print.h
lib.o: lib.c lib.h
print.o: print.c print.h
```

`make` sees, in place of line 11:

```
main.o: main.c lib.h print.h
lib.o: lib.c lib.h
print.o: print.c print.h
```

```
project/
├── Makefile
├── app
├── lib.c
├── lib.d          built
├── lib.h
├── lib.o
├── main.c
├── main.d         built
├── main.o
├── print.c
├── print.d        built
├── print.h
└── print.o
```

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

`make` Cheat Sheet

command	what it does
`make`	build the first target
`make print.o`	build the target `print.o`
`make clean`	build the target `clean`
`make -n`	only **show** the commands (dry run)
`make -B`	build everything again
`make -j8`	run up to 8 recipes at the same time
`make -C dir`	use the `Makefile` from `dir`
`make CC=clang`	change a variable
`make --debug=v`	show every decision, indented by the target stack

Remember

- a target is rebuilt when it is **missing** or **older** than a prerequisite
- `$@` the target, `$<` the first prerequisite, `$^` all of them
- a header goes on the rule of the `.o` of every `.c` that includes it, never on the executable
- `make` compares **modification times**, it does not read the code
- targets that are not files (`all`, `run`, `clean`) go in `.PHONY`

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

Version Control with Git

Save points for your code

Bibliography

For this section

1. **Scott Chacon, Ben Straub**, *Pro Git*, 2nd Edition, Apress, 2014, online
   - Chapter 1 - *Getting Started*
     - Section 1.1 - *About Version Control*
     - Section 1.3 - *What is Git?*
   - Chapter 2 - *Git Basics*
     - Sections 2.1 - 2.3, 2.5
   - Chapter 3 - *Git Branching*
     - Section 3.1 - *Branches in a Nutshell*

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

What Is Git

A program that remembers every version of a project

- a **version control system**: keeps the *history* of the files
- go back to any version, see **who** changed **what** and **why**
- many people work on the **same code** at the same time
- **distributed**: every copy has the whole history
- written in 2005 by *Linus Torvalds*, for Linux

❌ **Without git**

```
Desktop/
├── project.zip
├── project_final.zip
└── project_final_v2.zip
```

💡 the folder with the files and their history is a **repository** (*repo*)

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

What Is Git

A program that remembers every version of a project

- a **version control system**: keeps the *history* of the files
- go back to any version, see **who** changed **what** and **why**
- many people work on the **same code** at the same time
- **distributed**: every copy has the whole history
- written in 2005 by *Linus Torvalds*, for Linux

❌ **Without git**

```
Desktop/
├── project.zip
├── project_final.zip
└── project_final_v2.zip
```

✅ **With git**

```
project/
├── .git/          the whole history
├── Makefile
└── main.c
```

💡 the folder with the files and their history is a **repository** (*repo*)

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

What Is Git

A program that remembers every version of a project

- a **version control system**: keeps the *history* of the files
- go back to any version, see **who** changed **what** and **why**
- many people work on the **same code** at the same time
- **distributed**: every copy has the whole history
- written in 2005 by *Linus Torvalds*, for Linux

❌ **Without git**

```
Desktop/
├── project.zip
├── project_final.zip
└── main.c
```

Sharing a repository

☁️ websites that keep git repositories, for free

🌐 *GitHub*, *GitLab*, *Codeberg*, *Bitbucket*, ...

⚠️ git ≠ GitHub: git works without them

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

Commits

A commit is a snapshot of the whole project, with a message

- a **commit** saves the state of **all** the files
- its id is the **hash**: `7cbb829`
- it has an author, a date and a **message**
- it points to its **parent**, the commit before it

```
project/
└── main.c    new file
```




```
$ git add main.c
$ git commit -m "Add main.c"
```

💡 like a save point in a game: you can always come back to it

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

Commits

A commit is a snapshot of the whole project, with a message

- a **commit** saves the state of **all** the files
- its id is the **hash**: `7cbb829`
- it has an author, a date and a **message**
- it points to its **parent**, the commit before it

```
project/
├── Makefile    new file
└── main.c
```




```
$ git add Makefile
$ git commit -m "Add the Makefile"
```

💡 the arrow goes from a commit to its parent

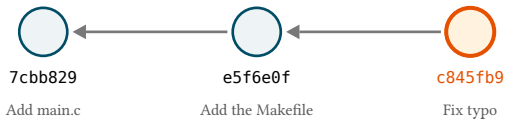
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Commits

A commit is a snapshot of the whole project, with a message

- a **commit** saves the state of **all** the files
- its id is the **hash**: `7cbb829`
- it has an author, a date and a **message**
- it points to its **parent**, the commit before it

```
project/
├── Makefile
└── main.c        edited
```




```
$ git add main.c
$ git commit -m "Fix typo"
```

💡 the chain of commits is the **history**: `git log --oneline` shows it

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

From the Editor to the Server

A change goes through four places



- you **edit** the files

```
(edit main.c in the editor)
```

```
project/
├── Makefile
└── main.c    edited
```

💡 the first three places are on **your computer**; `git status` shows what is in each

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

From the Editor to the Server

A change goes through four places



- you **edit** the files
- `git add`: pick what goes in the commit

```
$ git add main.c
```

```
project/
├── Makefile
└── main.c      staged
```

💡 the first three places are on **your computer**; `git status` shows what is in each

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

From the Editor to the Server

A change goes through four places



- you **edit** the files
- `git add`: pick what goes in the commit
- `git commit`: save in the **local** repository

```
$ git commit -m "Fix typo"
```

```
project/
├── Makefile
└── main.c        committed
```

💡 the first three places are on **your computer**; `git status` shows what is in each

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

From the Editor to the Server

A change goes through four places



- you **edit** the files
- `git add`: pick what goes in the commit
- `git commit`: save in the **local** repository
- `git push`: send to the **remote**

```
$ git push
```

```
project/
├── Makefile
└── main.c      pushed
```

💡 the remote is **shared**: now your colleagues can get your commit

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

From the Editor to the Server

A change goes through four places



- you **edit** the files
- `git add`: pick what goes in the commit
- `git commit`: save in the **local** repository
- `git push`: send to the **remote**
- `git pull`: get the others' commits

```
$ git pull
Already up to date.
```

```
project/
├── Makefile
└── main.c      pushed
```

💡 `git pull` brings the commits your colleagues pushed; here there are none yet

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

Commit vs Push

A commit stays on your computer until you push it

`git commit`

- saves a snapshot in the **local** repository
- works **offline**, only you see it

`git push`

- sends the commits to the **remote**
- now the others can `git pull` them

```
$ git add main.c
$ git commit -m "Fix typo"
[main c845fb9] Fix typo
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
```



💡 *ahead by 1 commit*: `c845fb9` is only on your computer, the server does not have it

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

Commit vs Push

A commit stays on your computer until you push it

`git commit`

- saves a snapshot in the **local** repository
- works **offline**, only you see it

`git push`

- sends the commits to the **remote**
- now the others can `git pull` them

```
$ git add main.c
$ git commit -m "Fix typo"
[main c845fb9] Fix typo
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
$ git push
To github.com:alice/project.git
   e5f6e0f..c845fb9  main -> main
```



💡 `e5f6e0f..c845fb9`: the server moved `main` to your commit

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

Commit vs Push

A commit stays on your computer until you push it

`git commit`

- saves a snapshot in the **local** repository
- works **offline**, only you see it

`git push`

- sends the commits
- now the others car

```
$ git add main.c
$ git commit -m "Fix typo"
[main c845fb9] Fix typo
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
On branch main
```

Commit vs Push

💾 **commit** = save, on your computer

📤 **push** = share, on the server

⚠️ not pushed: your colleagues do not see it

```
Makefile
main.c        pushed
```

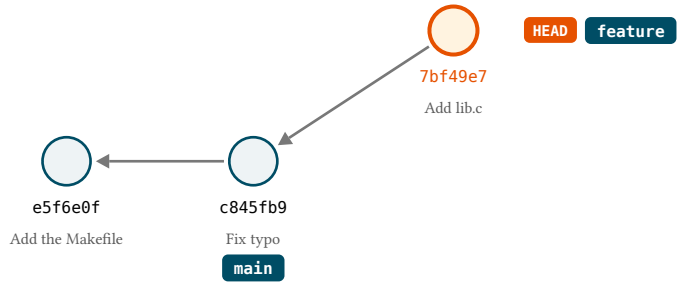
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Branches

A branch is a name that points to a commit

- a **branch** is a label that moves
- `HEAD`: the branch you are on
- a commit moves **this** branch
- work on a branch, `main` stays safe
- **merge** brings the work into `main`

```
project/
├── Makefile
├── lib.c      new file
├── lib.h      new file
└── main.c
```




```
$ git switch -c feature
$ git add lib.c lib.h
$ git commit -m "Add lib.c"
```

💡 `git switch -c` makes a branch and moves `HEAD` to it

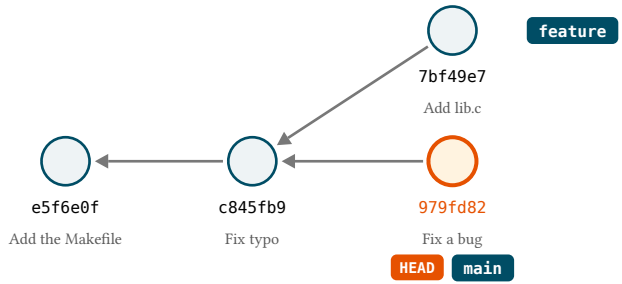
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Branches

A branch is a name that points to a commit

- a **branch** is a label that moves
- `HEAD`: the branch you are on
- a commit moves **this** branch
- work on a branch, `main` stays safe
- **merge** brings the work into `main`

```
project/
├── Makefile
└── main.c        edited
```




```
$ git switch main
$ git commit -am "Fix a bug"
```

💡 back on `main`, `lib.c` and `lib.h` are not in the folder

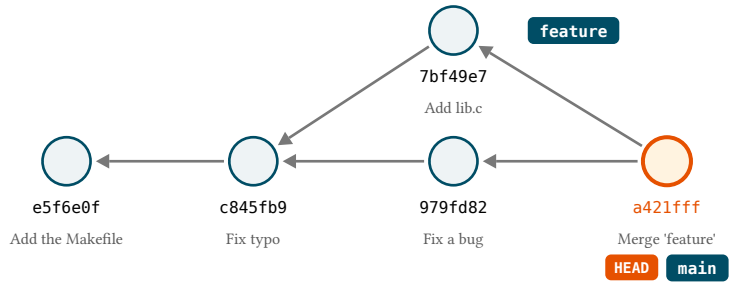
 Copyright (C) 2026 Wyliodrin & Politehnica Bucharest, CC-BY-SA 4.0

Branches

A branch is a name that points to a commit

- a **branch** is a label that moves
- `HEAD`: the branch you are on
- a commit moves **this** branch
- work on a branch, `main` stays safe
- **merge** brings the work into `main`

```
project/
├── Makefile
├── lib.c      merged
├── lib.h      merged
└── main.c
```




```
$ git merge feature
```

💡 the merge commit has two parents; `main` now has `lib.c` too

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

A Few Commands

Enough for a project

command	what it does
Start	
`git clone <url>`	copy a remote repository
`git init`	new repository, here
Every day	
`git status`	what changed, what next
`git add <file>`	stage a change
`git commit -m "..."`	save a snapshot
`git push`	send the commits
`git pull`	get the others' commits

command	what it does
Look	
`git log --oneline`	the history
`git diff`	changes not staged yet
Branches	
`git branch`	list the branches
`git switch -c <name>`	new branch, go to it
`git switch <name>`	go to a branch
`git merge <name>`	merge it into this branch

💡 `git status` is always safe: it tells you what to do next

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

From an Editor to an IDE

Language servers + debug adapters

Bibliography

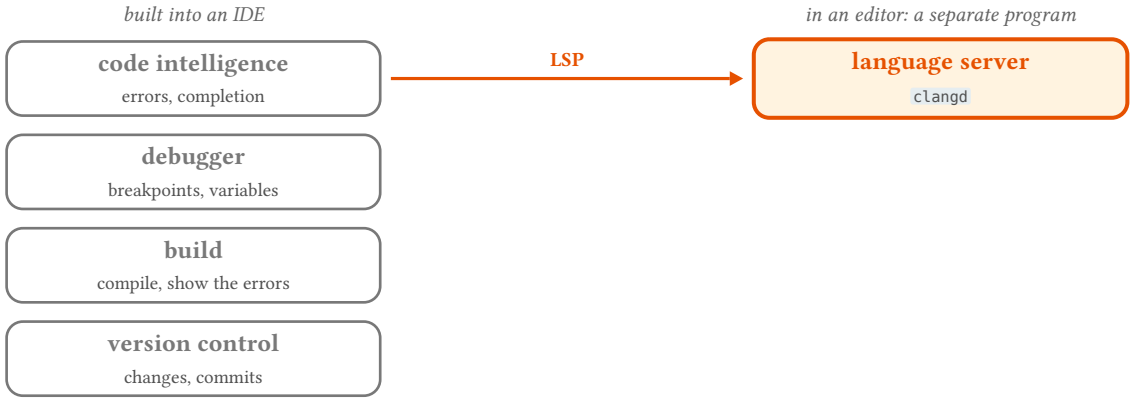
For this section

1. *Language Server Protocol Specification*, online
2. *Debug Adapter Protocol Specification*, online
3. *VSCodium / VS Code documentation*, *Tasks*, *Debugging*, online

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

Editor + Plugins = IDE

Every part of an IDE is a separate program, the editor only shows the results

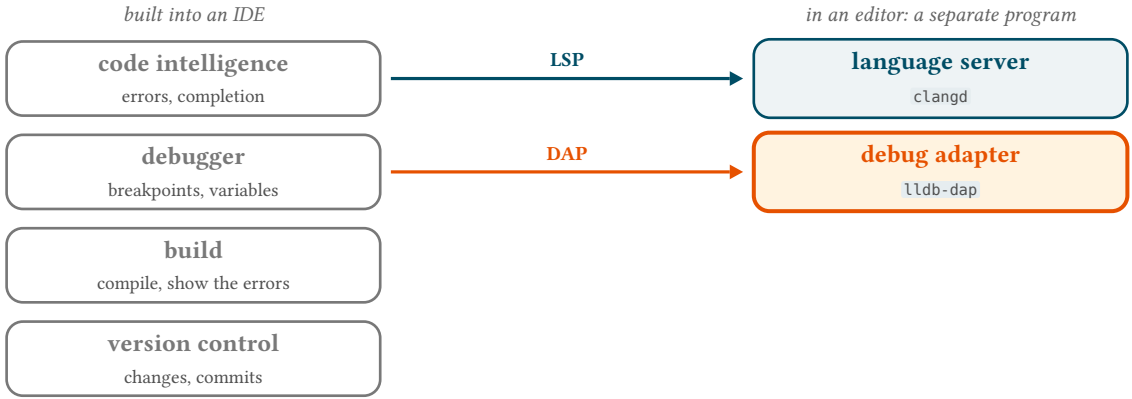


💡 the **language server** understands the code, the editor asks it through LSP

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

Editor + Plugins = IDE

Every part of an IDE is a separate program, the editor only shows the results

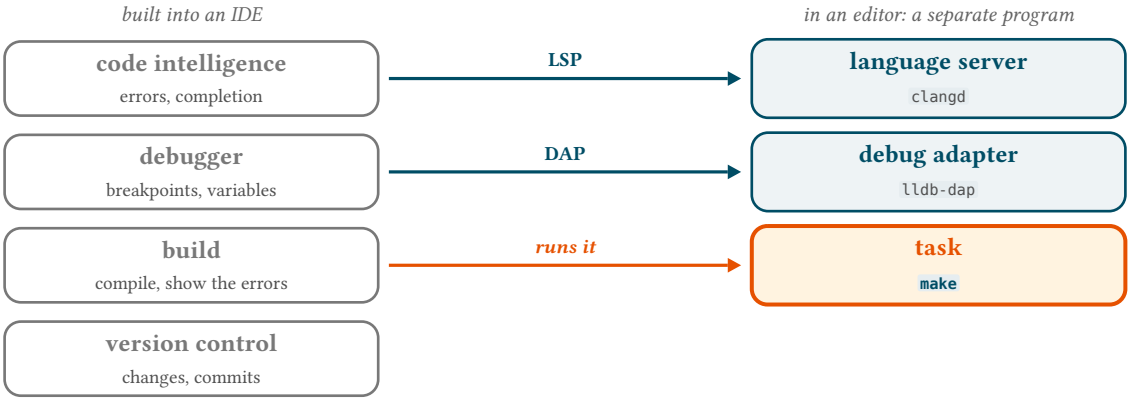


💡 the **debug adapter** drives the debugger, the editor asks it through DAP

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

Editor + Plugins = IDE

Every part of an IDE is a separate program, the editor only shows the results

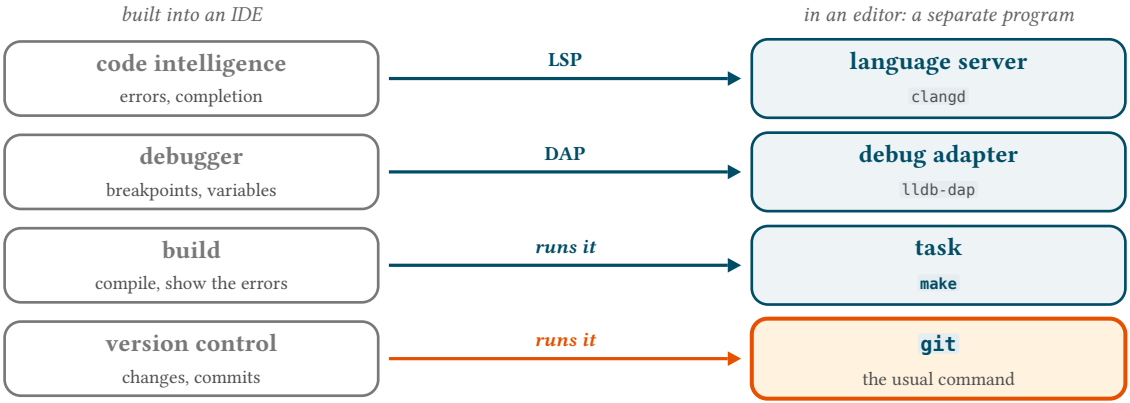


💡 a **task** runs the build, the editor reads the errors from its output

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

Editor + Plugins = IDE

Every part of an IDE is a separate program, the editor only shows the results



💡 the editor runs `git` and shows the changes next to the code

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

Configure the Project

Two files tell the editor how to build and how to debug

Build: a *task*

`.vscode/tasks.json`

```
{
  "version": "2.0.0",
  "tasks": [{
    "label": "make",
    "type": "shell",
    "command": "make",
    "problemMatcher": ["$gcc"]
  }]
}
```

💡 a task has a name, its `label`; `problemMatcher`: the editor reads the `gcc` errors

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

Configure the Project

Two files tell the editor how to build and how to debug

Build: a *task*

`.vscode/tasks.json`

```
{
  "version": "2.0.0",
  "tasks": [{
    "label": "make",
    "type": "shell",
    "command": "make",
    "problemMatcher": ["$gcc"]
  }]
}
```

Debug: a *launch configuration*

`.vscode/launch.json`

```
{
  "version": "0.2.0",
  "configurations": [{
    "type": "lldb",
    "request": "launch",
    "name": "Debug app",
    "program": "${workspaceFolder}/app",
    "preLaunchTask": "make"
  }]
}
```

`launch.json`: `"preLaunchTask": "make"` *runs the task* `"label": "make"` : `tasks.json`

💡 F5: first the **`make`** task builds `app`, then the debugger starts it

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

Configure the Project

Two files tell the editor how to build and how to debug

Build: a *task*

`.vscode/tasks.json`

```
{
  "version": "2.0
  "tasks": [{
    "label": "mak
    "type": "shel
    "command": "m
    "problemMatch
  }]
}
```

Debug: a *launch configuration*

`.vscode/launch.json`

📂 **`.vscode/launch.json` is read by**

VS Code, VSCodium	its own file
Zed	if there is no `.zed/debug.json`
`neovim` + `nvim-dap`	the configurations, but not `preLaunchTask`: run **`make`** yourself first

`launch.json`: `"preLaunchTask": "make"` *runs the task* `"label": "make"` : `tasks.json`

💡 `launch.json` is not only for VSCodium

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

Editor Settings in the Project

The settings of one editor, saved inside the project

- a **hidden** folder: its name starts with `.`
- in the root folder of the project
- only for **this** project and **this** editor

editor	folder
VS Code, VSCodium	`.vscode/`
Zed	`.zed/`
`helix`	`.helix/`
JetBrains IDEs	`.idea/`
`vim`, `neovim`	a file: `.exrc`, `.nvim.lua`[1]

```
project/
  .vscode/  VSCodium
    tasks.json  build
    launch.json  debug
  .zed/  Zed
    debug.json  debug
  Makefile
  main.c  lib.c  lib.h  ...
```

💡 commit it with `git`: the same build and debug setup for everyone

[1] read only if the `exrc` option is on; `neovim` asks first if you trust the file. `nvim-dap` can also read `.vscode/launch.json`

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

LSP and DAP in Editors

Most editors speak both protocols

editor	language servers (LSP)	debug adapters (DAP)
VS Code / VSCodium	an extension per language	an extension per debugger
Zed	built in	built in
`helix`	built in	built in, experimental
`neovim`	built in	plugin: `nvim-dap`
`emacs`	built in: `eglot`	package: `dape`
`vim`	plugin: `vim-lsp`	plugin: `vimspector`

💡 *built in*: the editor speaks the protocol, you only install the server or the adapter

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

LSP and DAP in Editors

Most editors speak both protocols

editor	language servers (LSP)	debug adapters (DAP)
VS Code / VSCodium	an extension per language	an extension per debugger
Zed	built in	built in
`helix`	built in	built in, experimental
`neovim`	built in	plugin: `nvim-dap`
`emacs`	built in: `eglot`	package: `dape`
`vim`	plugin: `vim-lsp`	plugin: `vimspector`

install `clangd` and `lldb-dap` once, then use them from **any** editor you like

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

Trust the Folder?

An IDE does not only show the code, it **runs** programs from the folder

Trust this folder?

`~/Downloads/homework`

The editor can run **language servers**, **build scripts**, **tasks** and the **debugger** from this folder. Only trust code you know.

✅ Restricted mode | ⚠️ Trust

VS Code, *Zed*, *JetBrains* IDEs, ... all ask this when they open a new folder

⚠️ Trust: dangerous for unknown code

- starts the **language server** and the programs the folder's **settings** name
- a **Rust** project: `rust-analyzer` compiles and **runs** `build.rs` as soon as the folder opens, **any code**, as you
- runs the **tasks** and the **debugger** configurations

✅ Restricted mode: safe

- only an editor: you can read and edit the code
- **nothing** from the folder runs, until you trust it

💡 code from the internet or from a colleague: open it in **Restricted mode**, trust only code you know

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

What Does Trust Allow?

A trusted folder decides which programs the editor **runs**

in the folder	the editor runs	when
`.vscode/settings.json` `.zed/settings.json`	the language server written there, with its arguments	**on open**

💡 the settings choose **which program** starts as the language server: any program can be named there

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

What Does Trust Allow?

A trusted folder decides which programs the editor **runs**

in the folder	the editor runs	when
`.vscode/settings.json` `.zed/settings.json`	the language server written there, with its arguments	**on open**
`Cargo.toml` `build.rs`	`rust-analyzer` runs `cargo check`, which compiles and **runs** `build.rs`	**on open**

💡 to understand a Rust project, `rust-analyzer` builds it, and building means **running** `build.rs`

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

What Does Trust Allow?

A trusted folder decides which programs the editor **runs**

in the folder	the editor runs	when
`.vscode/settings.json` `.zed/settings.json`	the language server written there, with its arguments	**on open**
`Cargo.toml` `build.rs`	`rust-analyzer` runs `cargo check`, which compiles and **runs** `build.rs`	**on open**

`.vscode/tasks.json`
`.vscode/launch.json`
`.zed/tasks.json`
	the commands written there	when you run a task or debug

💡 *Run* and *Debug* do not only start your program: they run whatever command the folder wrote

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

What Does Trust Allow?

A trusted folder decides which programs the editor **runs**

in the folder	the editor runs	when	an attacker writes
`.vscode/settings.json` `.zed/settings.json`	the language server written there, with its arguments	**on open**	`"path": "./x.sh"`
`Cargo.toml` `build.rs`	`rust-analyzer` runs `cargo check`, which compiles and **runs** `build.rs`	**on open**	`build.rs`: any Rust code

`.vscode/tasks.json`
`.vscode/launch.json`
`.zed/tasks.json`
	the commands written there	when you run a task or debug	`"command": "rm -rf ~"`

⚠️ **on open**: nothing to click, the code runs **as you**, with your files and your SSH keys

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

We talked about

- Compiled & Interpreted Languages
  - compiler, interpreter, bytecode
- Integrated Development Environments
- Language Servers
- Debuggers
- Debug Adapters
- The C Compilation Pipeline
  - preprocessor, compiler, assembler, linker
- Building with `make`
  - rules, step by step
- A `Makefile` for a Project
  - automatic variables, pattern rules
- Special Targets
  - `.PHONY`, `clean`, `all`, `run`
- Header Dependencies
- Version Control with Git
  - commits, commit vs push, branches
- From an Editor to an IDE
  - LSP + DAP

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