Tooling and workflow

Raul P. Pelaez

September 12, 2025

Contents

1 Introduction

In this first lesson, we will cover the tools we will be using throughout the course, as well as the typical workflow we will follow to start developing code or run it.

Info

About these notes
I use these notes to organize my ideas and structure the lessons. It is my intention, during each lesson, to cover everything in the notes and nothing else. I constantly improve on these notes in terms of format and content according to your feedback, so please do not hessitate to contact me if you see something incorrect or lacking!

Insight

The messiness of software distribution
Modern development requires a lot of extremely sophisticated tools working together. Each *environment* has its own set of tools, and each tool has its own set of dependencies. One of the soft skills I want you to take away from this course is the ability to navigate this messiness. In the case of C, being such an essential component in all operative systems, we could probably get away with using the already-present system compiler and a basic text editor. However, learning how to use a terminal, a package manager, a code editor and a build system will allow you to work with any language and any project, no matter how complex it is. In any case, I tried to keep the tools to a minimum, and to streamline the installation and workflow as much as possible. You will see, though, that even so, complexity always creeps in and your computer will complain in obscure and unexpected ways. This is normal, and it is part of the process of learning how to develop software. The important thing is to learn how to deal with these issues, and how to find the information you need to solve them. I will try to help you with this, but ultimately it is up to you to learn how to navigate this messiness.

Advice

We cannot make simple software
Software is inherently complex. This complexity is made evident to us day to day when interacting with our computers, phones, cars, etc. Bugs and quirks appear constantly. Perhaps a program will crash, a button will not work, your car window will only open halfway, or your phone will not connect to the wifi. As developers, we must strive to make our software as robust and reliable as possible, but even so, we cannot control most of the software we use day to day. The sad reality of things is that there is only one way to go about it: Deal with it. This soft skill is one of the most important ones to learn, and it is not easy. It requires patience, perseverance, and a godly immunity to frustration. I leave you here some resources that I found interesting about this:

- Why Can't We Make Simple Software?
- Everything is broken: One week of bugs
- Software disenchantment

Regardless of the language of choice, we will need to use a set of tools that fill specific roles in the development process. Sometimes, a single tool provides several of these roles. Other times, even when a tool provides a solution for something, we will prefer to use a different tool for that role, because it is more convenient or because it is the standard in the community. The kind of tools we will need are:

- Terminal emulator, to interact with the operating system.
- Package/environment manager, to install and manage the dependencies of our project.
- Code editor, to write and edit our code.
- Build/deploy system, to compile and link our code.
- Version control system, to keep track of the changes in our code and collaborate with others.
- Other software engineering tools, such as linters, formatters, documentation generators, testing frameworks, etc.

In this course, we will use the following tools to cover these roles:

- **Terminal emulator**: We will use the terminal that comes with your operating system; PowerShell in Windows, Terminal in macOS, and the terminal in Linux. We will also use the terminal integrated in the VSCode editor.
- **Package/environment manager**: We will use `conda` to manage the dependencies of our projects, a tool that allows us to create isolated environments with specific versions of the dependencies we need.
- **Code editor**: We will use the VSCode editor, which is multi-platform and multilingual.
- **Build/deploy system**: We will use CMake, a standard build system that allows us to compile and link our code in a platform-independent way.
- **Version control system**: We will use git, the de-facto standard.

We chose these tools because they are *standard* in the industry, and they are well-supported and documented. They are also easy to install and use in any OS, and they have a large community of users

that can help us when we run into problems. Finally, we specifically chose them because they *talk* to each other automagically. For instance, the VSCode editor has built-in support for git, and it can automatically detect the `conda` environment we are using. It will also recognize the CMake files in our code and use it to provide us with features such as code completion, syntax highlighting, and error checking.

Info

Useful online resources
A general word of advice: Be extremely skeptic about online information. Most resources you will find in the wild will be outdated, wrong, or just plain bad. Detecting good resources is a skill that you will develop with time. In a similar note, be very critical of code or advice generated by LLMs, which have been trained in part with all these terrible resources.

- Compiler Explorer: An online tool to see the assembly code generated by different compilers for a given piece of C/C++ code. We can use it for quick tests, explore assembly, debug compilation issues, etc.
- C reference: The de-facto standard reference for the C (and C++) language and its standard library.
- C template project: A minimal C project template with CMake, conda environment file, gitignore, and a few other things. You can use it as a starting point for your own projects or homework. I wrote it specifically for this course.

Lets take a look at each of these tools in more detail, and how we will use them in this course.

2 Using the terminal

The commandline (we will also call it the *terminal* or *shell*) is a text-based interface to the computer. The terminal is the most efficient way of interfacing with a computer, also allowing for more complex operations than a graphical interface. In some cases, the terminal is the only way to interact with a computer, like when working with remote servers or embedded systems.

The terminal has a steep learning curve, but it is well worth the effort.

Advice

Being used to graphical interfaces, using the terminal might seem like a step back at first. You will be slower and clumsy. You might find it unnecessarily complicated. This is normal, but with practice you will become faster and more efficient than you ever were with a mouse.

Terminals and Operative Systems

Each OS has its own terminal. Windows has the Command Prompt and PowerShell, MacOS has the Terminal app, and Linux has a variety of terminals, like GNOME Terminal, Konsole, and xterm.

The program that runs in the terminal to receive our commands is called a shell. The default shell in Linux is usually bash, while MacOS uses zsh. Windows uses cmd.exe or PowerShell (confusing, I know). If you are using Windows, ignore the existence of cmd.exe and use PowerShell instead.

Info

Our basic workflow will consist on the following steps:

- Open a terminal.
- Navigate to a folder for the course: `cd ~/path/to/course`
- Create a new folder for the lesson: `mkdir lesson1`
- Navigate to the lesson folder: `cd lesson1`
- Activate the Conda environment: `conda activate my_env`
- Open VSCode to edit the project: `code .`
- Build from the terminal: `cmake -B build && cmake --build build`
- Run the program: `./build/my_program`

We will use the terminal to build and run C projects and to interact with the filesystem. We will also use it to install packages and to open sources for editing.
Do not worry, by the end of these notes, you will be introduced to all these tools. Keep reading.

Info

Opening the terminal
Each OS has a different way of opening the terminal.

- In Windows, you can press `Win+R`, type `PowerShell` and press `Enter`. Alternatively, you can search for `PowerShell` in the Start menu.*
- In MacOS, you can press `Cmd+Space`, type `Terminal` and press `Enter`.
- In Linux, you can press `Ctrl+Alt+T` to open a terminal.

*Note that in Windows, should you choose to go the "Windows way", you will always open "Anaconda Powershell" instead, keep reading.

How the terminal looks

When you open a terminal, you will be presented with something similar to the following: In Windows, you might see:

```
(base) PS C:\Users\raul>
```

In OSX:

```
(base) raul@MacBook ~ %
```

Lets break it down:

- `(base)`: The currently activated conda environment. If you cannot see this, you did not install conda correctly.
- `PS`: The shell you are using. In this case, PowerShell.
- `C:\Users\raul`: The current working directory. In this case, the user `raul`'s home directory.
- `>`: The prompt symbol in PowerShell. In OSX, it is a `%`.
- `~`: In OSX, the tilde means "the home directory".
- `raul@MacBook`: The user and the computer name in OSX.

We can input commands into this prompt and the terminal will run them.

2.1 Basic commands

Info

Moving around
When opening your terminal, you will be placed in your HOME directory. This is your current working directory. You can move around the filesystem using the following commands:

- `cd`: Change directory. Use `cd folder` to enter a folder, `cd ..` to go up one level.
- `ls`: List files in the current directory.
- `pwd`: Print the current working directory.

The `.` and `..` directories

- `.`: Refers to the current directory.
- `..`: Refers to the parent directory (the directory above the current one).
- `~`: Refers to the home directory of the current user.

For instance, going to the parent directory of the current one can be done with `cd ..`, and going to the home directory can be done with `cd ~`.

Info

Creating and deleting files and directories

- `mkdir`: Make a new directory.
- `rmdir`: Remove an empty directory.
- `rm`: Remove a file. Use with caution.
- `rm -r`: Remove a directory and all its contents. Use with caution.

Info

Working with files

- `mv`: Move or rename a file. Use `mv file1.txt file2.txt` to rename a file, or `mv file1.txt folder/` to move a file to a folder.
- `cp`: Copy a file. Use `cp file1.txt file2.txt` to copy a file, or `cp file1.txt folder/` to copy a file to a folder

You can press `TAB` to autocomplete commands and file names. This is a huge time saver, as you will not have to type the full name of a file or command. Try to press `TAB` several times. The `history` command will show you a list of the commands you have run in the terminal. You can also use the up and down arrows to navigate this list and rerun commands.

Curiosity

The UNIX heritage
I chose these commands because all OSs will recognize them, which happens not by chance! They originated in early UNIX systems in the late 1960s and 1970s, when Ken Thompson, Dennis Ritchie (yes, the creator of C), and others designed UNIX at Bell Labs. UNIX emphasized a small set of simple, composable commands that could be combined with pipes and redirection. Many of those core utilities, ls (list directory contents), cd (change directory), mkdir (make directory), rm (remove), cp (copy), etc. were introduced then.
That design philosophy was so influential that these commands (or close equivalents) were carried over into later UNIX-like systems, such as BSD, Linux, and macOS. Even Windows eventually adopted many of them in PowerShell and WSL to improve interoperability with UNIX-style environments.

3 For Windows users

If you have a Mac, Linux, a WSL installation or you have a Windows with a Virtual Machine running Linux, you can skip this section.

Warning

You must make a choice
It is my firm believe that one that wants to call themself a Computer Scientist should learn (and suffer through) to use their hardware to the fullest. Whatever it is that you have at hand. Some of you have Windows computers, and that is fine. However, Windows is not a developer-friendly OS. It presents extremely idiosyncratic behaviors that are by no means easy to overcome for a newcomer.
In my opinion, the pain you will have to endure to learn to develop C in Windows far exceeds the level of frustration that I can justify in course like this. Still, I value your freedom of choice, your curiosity, and your willingness to learn. Thus, if you wish to develop code "the Windows way" I will try my best to support you and help you troubleshoot.
Having said this, fret not, for I will give you some options to make your life easier even if you use Windows as your main OS. Keep reading this section.

Being a Windows user, these are your options to develop C (or almost anything else, really) code:

- Embrace the Windows way: Install Visual Studio (not to be confused with VS *Code*) with the Windows SDK and the C/C++ developer kit. Install Anaconda and use the Powershell.
- **(Recommended)** Use WSL (Windows Subsystem for Linux): Install WSL with a Linux distribution of your choice (I recommend Ubuntu). You will have access to a full Linux terminal, and you can install all the tools we will use in this course. You can also access your Windows files from WSL, so you can use VSCode in Windows to edit your code.
- Dual boot your computer with Linux (Fedora or Ubuntu). If you are curious to explore Linux (I really think you should at some point in your compsci journey), go ahead and dual boot. It will be overkill for this course, though, and you will most probably bork your computer a couple times in the process.
- Use a Virtual Machine (VM) with Linux inside Windows. This option is quite heavyweight and the performance will be subpar. I do not recommend this option for you.
- Use the GNU stack with MinGW via the MSYS2 distribution. This is native, but not really "the Windows way".

Insight

It really saddens me to low-key send you the message that the way to develop code in Windows is to, essentially, not use Windows. However, this is the reality of things.

Advice

Be pragmatic: Choose WSL
This is the most lightweight, easy to install option. It will allow you to follow the course without issues. The only downside is that your applications will not be native Windows apps, rather will have to run inside WSL. In practice, this is not a big deal and will not affect you in any capacity.

I will provide guidance for two of these options:

3.1 The Windows way

Developing native code using the tools Microsoft provides is a bit of a nightmarish experience, but you can get something working in a mostly straightforward way if you do not mind installing around 15 GB of software. You will need to follow these (high-level) steps:

- Install Visual Studio using this link. During the installation, make sure to select the "Desktop development with C++" workload. This will install the necessary C/C++ compiler and tools.
- Install the Anaconda conda distribution using this link instead of Miniforge (I discuss `conda` in the next section). Works better in PowerShell.
- Install VSCode by choosing the "System Installer" option from this link.

After installing all of the above, whenever you want to start developing, hit the Windows key and type "Anaconda", select "Anaconda Powershell". From there, you can navigate to your project using `cd` and

friends. You can open VSCode in the current directory with `code .`, and you can build your project with CMake as explained later. However, you will need to call the CMake command with an additional argument to specify the generator, as follows:

```
(cppdev) PS > cmake -G "Visual Studio 2022" -B build
(cppdev) PS > cmake --build build
```

Even so, you will find that some of the code we will cover needs some tweaking to work in Windows. Although I will try to keep it to a minimum, the MSVC compiler is not fully compilant with the latest C standards. Thus, it will refuse to compile some "legal" code.

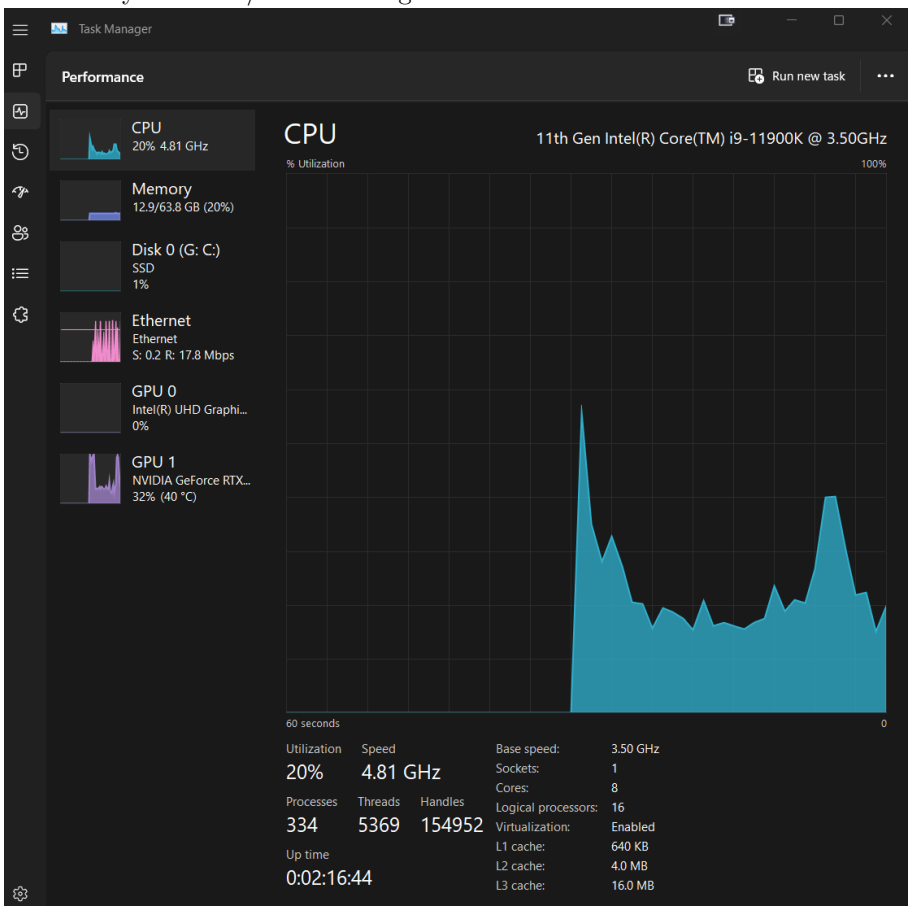
3.2 The UNIX way (WSL)

This is the option I recommend. It is straightforward to install, it is supported by Microsoft and you will have access to a full Linux terminal.

The Windows Subsystem for Linux (WSL) is a compatibility layer for running Linux binary executables natively on Windows 10 and Windows 11. It allows you to run a Linux distribution (such as Ubuntu, Debian, Fedora, etc.) directly on Windows without the need for a virtual machine or dual booting. See its official documentation here.

To install WSL, follow these steps (see here if you experience issues):

- Make sure Virtualization is enabled. You can check this in the Task Manager under the "Performance" tab. You should see "Virtualization: Enabled" in the bottom right corner. If it is disabled, you will need to enable it in your BIOS/UEFI settings. Follow the instructions here.



- Open PowerShell as Administrator (right-click the Start button and select "Windows PowerShell (Admin)").
- Type `wsl --install` and press Enter. This will enable the necessary features and install the default Linux distribution (usually Ubuntu). You might be prompted to restart your computer.
- Open PowerShell and type `wsl` to launch the Linux terminal. You will be prompted to create a new user and password for your Linux distribution. You can also access wsl by hitting the windows key

and typing "Bash" or "Ubuntu".
- Install Miniforge (a minimal conda distribution) by following the instructions in the next section.
- Install VSCode from this link.
- Open VSCode and install the "Remote - WSL" extension in VSCode. This will allow you to open folders in WSL directly from VSCode.
- Open PowerShell and type `wsl` to launch the Linux terminal. Navigate to some project folder using `cd` and friends. You can open VSCode in the current directory with `code .`, and you can build your project with CMake as explained later.
  I recorded a short video showing you how to follow these steps, you have it here. Please follow it carefully, as it is easy to miss a step.
  From now on, you can follow the instructions as if you were using Linux (or MacOS).

4 Conda: Virtual Environments and dependencies

C gives us the ability to write absolutely anything, and one of the pinacles of its adoption comes from its simplicity in terms of features and its standard library. However, we often do not want to write everything from scratch ourselves, relying instead on libraries that others have written. Another pillar of C's success is its status as the lingua franca of programming, which permits it to interface with libraries written not only in C, but in almost any other language. These libraries can be anything from a simple math library to a complex graphics engine, and they are often written in C or C++. Unfortunately, the simplicity of C stops when we open the door to external libraries. Each library we include in our project probably requires to use other libraries in turn (called *dependencies*), and so on. Furthermore, software is constantly evolving, and dependencies are often tied to specific versions of other dependencies. It is not enough to say that our project depends on `SDL`, we need to specify that it works for `SDL==2.26`.

You can see how installing these dependencies can quickly become a nightmare (mind you, this issue is not particular to C). This problem is solved by using a *package manager*, which is a tool that automates the process of installing, upgrading, configuring, and removing software packages. Every Linux distribution has its own package manager (such as `apt` or `dnf`), MacOS has `Homebrew` that is a de-facto standard, and Windows... well, there are some options out there.

There is another side to this problem. Say that you are working on two projects. One of them is kind of old and only be compiled with `gcc==4.9`, the other one includes some new features and can only be compiled with `gcc>12`. If you use your OS package manager to install `gcc==4.9`, you cannot compile the second project, and vice versa. This is a common problem in software development, and it is known as *dependency hell*. The solution to this problem is to use *virtual environments*, which are isolated environments that allow you to install and manage dependencies for each project separately. This way, you can have different versions of the same library installed in different environments, and you can switch between them easily.

Info

Package managers
Tools like `apt`, `dnf`, `brew` and `choco` are package managers that allow you to install, upgrade, and remove software packages in your system.

Info

Virtual environments
Isolated software environments that allow you to install and manage dependencies for each project separately. This way, you can have different versions of the same library installed in different environments, and you can switch between them easily.

In this course, we will use `conda`, which is both a package manager and an environment manager. It allows us to create isolated environments with specific versions of the dependencies we need, and it also allows us to install and manage packages in those environments. Conda is widely used in the community, and it is a standard for Python development. It is also available for Windows, MacOS, and Linux, making it a cross-platform solution.

Info

Conda
A package and environment manager. It is multiplatform (Windows, MacOS, Linux) and multilingual (Python, C, C++, R, Java, and more).

Info

Installing conda
We will use the Miniforge distribution (or Anaconda if you are on Windows). Installation takes just a few seconds, but it is important to follow the instructions carefully, as they are deceptively simple. I recorded a short video showing how to install it in Windows, MacOS and Linux: Installation of conda and VSCode. Please follow it carefully, as it is easy to miss a step.
Afterwards open a new terminal (remember to open the "Anaconda Powershell" if you are on Windows without WSL), you should see "(base)" at the beginning of the prompt. This means that you are in the base environment. You can now create a new environment with:

```
conda create --name myenv cmake clang
conda activate myenv
```

Your prompt will change from "(base)" to "(myenv)", signaling that the terminal is now in that environment. You will need to activate the environment every time you open a new terminal. Note that the environment is only in effect for the terminal you are using, so you can have different environments in different terminals. You can also install packages with conda install after creating an environment.

```
conda install jupyter
```

Never install packages in the base environment!

Advice

Naming of folders and files in a computer
Operative Systems **hate** spaces in file and folder names. They are a source of endless problems. Use underscores or camelCase instead. For instance, use `my_project` instead of `my project`. Spaces will break your scripts, conda environments, python imports, and many other things.
Additionally, avoid using special characters like `!@#$%^&*()` in file and folder names. One issue with special characters in names is that one has to *escape* them when using them in the terminal. For instance, if you have a folder named `my folder`, you have to write `my\ folder` in the terminal to refer to it, which is annoying and error-prone.

Advice

Environments are ephemeral
Environments are not meant to be permanent. They are meant to be created, used, and then discarded. You can delete an environment with:

```
conda env remove -n myenv -y
```

You can list all the environments you have with:

```
conda env list
```

It is, in general, a bad idea to install packages to an already existing environment. It is always better to create a new environment from scratch, adding or removing packages as needed using an environment file (see next section).

4.1 Environment files

In a project controlled by conda, we will have a file called `environment.yml` in the root directory. This file contains the list of dependencies that the project needs to run. It is a YAML file, which is a human-readable data serialization format. The `environment.yml` file is used to create a new conda environment with all the dependencies needed for the project. You can create a new environment from this file with:

```
conda env create -f environment.yml -n some_name
```

This will create a new environment called `some_name` with all the dependencies specified in the `environment.yml` file. If the environment file is called `environment.yml`, we can skip it in the above call. You can then activate the environment with:

```
conda activate some_name
```

This will change your prompt to show the name of the environment, so you know you are in that environment. You can also use the `conda env list` command to see a list of all the environments you have created, and which one is currently active.

An example of an `environment.yml` file is:

```yaml
name: my_project
channels:
   - conda-forge
   - nodefaults
dependencies:
   - cmake>=3.27
   - python
   - clang==18.1.8
```

This file specifies that the environment is called `my_project`, and it will use the `conda-forge` channel to install the dependencies (a community-mainained repository, leave it like that always). The dependencies are listed under the `dependencies` key, and in this case, we are installing `cmake`, `python` and `clang`. We can also specify the version (or range of versions) of each dependency we want to install. In this case, we are specifying that we want `cmake` version 3.27 or higher, and `clang` version 18.1.8 exactly.

Insight

We can, for the most part, write environment files that will work for any of the supported OSs. In practice, there are always quirks and unexpected setbacks in doing so. Each computer is a world of its own, and albeit learning to troubleshoot these kind of issues is a time consuming and frustrating process, it is my intention to force you a little bit into this chaos, as it is an essential skill for any developer.

5 VSCode: The code editor

The code editor is the tool we will use to write and edit our code. There are many code editors available, but we will use Visual Studio Code (VSCode) because it is free, open-source, and cross-platform. It is also very popular in the industry and integrates well with the other tools we will use in this course. Installing VSCode is straightforward, but I also included instructions in the conda installation video here.

Advice

AI first editors
VSCode has many AI-centered forks, such as Cursor. I strongly recommend you against using them as your main tool during your training stage. They can be useful in the hands of an experienced developer, but they will hinder your learning process. You will become dependent on them, and you will not learn how to solve problems on your own. They are a crutch that you should avoid using until you are comfortable with the basics of programming and software development.

Insight

The choice of an editor
I believe that freedom of choice in your tooling and how you use the computer is essential. I choose to stick with VSCode in this kind of course because of its simplicity and availability. It is a good editor for begginners. I strongly encourage you to explore other editors, be it during this course or later on. There are many others out there, such as Vim or Emacs (my personal choice). These alternatives, although extremely powerful in the long run, present a steep learning curve (we are talking about years of practice). At least during this course, however, keep VSCode installed and working, so you are not blocked.

6 CMake: The build system

C is a compiled language, meaning that the code we write needs to be transformed into machine code before it can be executed. This process is called *compilation*, and it is done by a program called a *compiler*. The compiler takes the source code we write and generates an executable file that can be run on the computer. At some point, our programs will grow beyond a single source code file, and we will need to compile and link several files together.

When your project grows it quickly becomes unfeasible to compile each file manually. Furthermore, the specific commands that will work for your system will probably not work for others. For this reason, it is common to write a script that will compile your project for you.

One of the most popular tools to do this is CMake. One describes the characteristics of the project in a file called CMakeLists.txt and then CMake will generate the necessary build files for your system. CMake is infamous for its obtuse syntax, thus we will try to keep it simple.

The typical workflow when encountering a CMake-enabled project is to run the following commands:

```
cmake -B build
cmake --build build
```

Warning

CMake in Windows
Native C development in Windows comes with a few quirks. For CMake, you will need to pass an additional argument for specifying a *generator*, which is the build system that CMake will use to generate the build files. The default generator in Windows is `NMake`, which is a command-line build tool that comes with Visual Studio and it is not installed by default. Read the section on Windows development for more details.

This will create a directory called build, enter it, run cmake to generate the build files and then run the build system to compile the project. Building will typically make some executable available under build/bin or somewhere around. It is also usual to "install" the project, which will copy the executable to some system folder where it can be run from anywhere. This is done by running the following command:

```
cmake --install .
```

Now any executable, shared library and/or header file that was installed can be used by any other project in the system as if it was a system library.

Advice

It is almost never a good idea to install your project in the system folders. Typically one works under a virtual environment (such as conda or a container) and installs the project there. CMake can be configured to install the project in a different folder:
`cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ..`
If you are using conda, the prefix would be `$CONDA_PREFIX` (written `$env:CONDA_PREFIX` in Windows).

6.1 Basic CMake syntax

CMake works by reading a file called `CMakeLists.txt`, which contains a set of commands that describe the project in a declarative way. Lets start with a simple example, in which we have the following project structure:

```
my_project/
|-- CMakeLists.txt
|-- library.c
|-- library.h
\-- main.c
```

The source codes are as follows:

`library.h`

```h
#ifndef LIBRARY_H
#define LIBRARY_H

int add(int a, int b);

#endif // LIBRARY_H
```

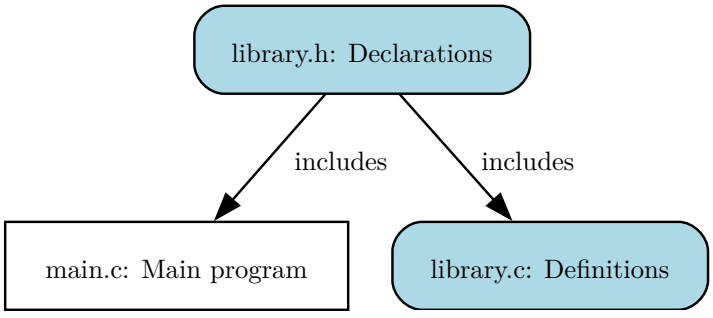
`library.c`

```c
#include "library.h"

int add(int a, int b) {
    return a + b;
}
```

`main.c`

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

int main() {
    int result = add(2, 3);
    printf("The result is: %d\n", result);
    return 0;
}
```

These files define a project with a library and an executable that uses it.



We could compile this project with the following commands:

```
clang -o library.o -c library.c
clang -o main.o -c main.c
clang -o my_program main.o library.o
```

However, this is tedious and error-prone. Instead, we can use CMake to automate this process. The `CMakeLists.txt` file for this project would be:

```
cmake_minimum_required(VERSION 3.15)
```

```
project(my_project VERSION 1.0 LANGUAGES C)
add_library(my_library library.c library.h)
add_executable(my_program main.c)
target_link_libraries(my_program PRIVATE my_library)
```

CMake asks for some basic metadata of the project. Then, we describe the nature and relations of the different sources (e.g, `library.c` is a library, `main.c` is an executable that links with `library`). In a more involved example, with a more complex directory structure, CMake allows to "split" this file by describing only the components in the directory where the `CMakeLists.txt` is located. Go to `https://github.com/RaulPPelaez/c-template-project` and inspect the top level `CMakeLists.txt`, notice how it "delegates" to the subdirectories by using `add_subdirectory`, go then to `src/CMakeLists.txt`, where you will see something similar to the above example.

7 Standards are important: Anatomy of a software project

Advice

Do not become overwhelmed
It is easy to become overwhelmed by the amount of tools and technologies available, and the sheer background knowledge necessary to fully understand and apply what you will find in this section. Do not worry if a large part of this section flies over your head. The general idea is enough for know. Project after project, you will encounter and adopt more and more of these tools. Give yourself a few years to learn them :P
Keep in mind, some of the technologies and tools mentioned here will not even be covered in this course!

Making your software project *expectable* helps the users in understanding it. In addition, following the community standards will make integration with tools seamless and mostly automatic. For instance, if the root of your project contains a file called `README.md`, repository-hosting sites like Github will render it as the main page of your project. If you have a file called `environment.yml`, `conda` will recognize it, and the VSCode editor will try to automatically enable it when editing something in that project. Naming your test files with a `test_` prefix will allow `pytest` (a Python testing framework) to automatically discover them, and so on and so forth.

The cognitive load of a new user will be much lower if they can expect to find a `README.md` file in the root of your project, or a `tests/` folder with the tests. If they see a file called `CMakeLists.txt` they can instantly guess that your project is installed with `cmake -B build && cmake --build build && cmake --install build`. A `docs/` folder will signal that documentation of your project is there, and so on and so forth. Below is an example of a typical C project structure skeleton that follows these conventions:

```
my_project/
|-- src/
|   CMakeLists.txt
|   \-- library.c
|   \-- internal_utils.c
|   \-- ... other source codes ...
|-- include/
|   my_project/
|   \-- library.h
|-- tests/
|   \-- test_something.c
|-- docs/
|   \-- ... documentation files ...
|-- .gitignore
|-- environment.yml
|-- CMakeLists.txt
|-- README.md
|-- .github/workflows/
```

```
    \-- ci.yml
```

The `include/` folder contains the public header files of the project, which are used to expose the public API of the library. The `src/` folder contains the source code of the project, implementing the API. The `tests/` folder contains the tests for the project, and the presence of a `CMakeLists.txt` file signals that we can compile and run them with the `ctest` command. . The `docs/` folder contains the documentation, the principal knowledge trove of the project besides the `README.md` file, which is the "frontpage" of the project. The `.gitignore` file tells git which files and folders to ignore when committing changes. Finally, the `.github/workflows/ci.yml` file contains the configuration for the continuous integration service, which might for instance try to build the project, run the tests and generate the documentation site automatically, ensuring that everything works as expected.

Advanced

Continuous integration (CI) & continuous deployment (CD)
Continuous integration (CI) and continuous deployment (CD) are practices that allow us to automate the process of building, testing and deploying our software. This is done by using a CI/CD service, such as GitHub Actions, Travis CI, CircleCI, etc. These services allow us to define a set of steps that will be executed automatically when we push changes to our code repository. This way, we can ensure that our code is always in a working state, and that it is always tested and deployed correctly.

When a user encounters this project, they can infer how to get its dependencies and install it (using `conda` and `cmake`), test it (using `ctest`). They can also expect to find the source code in the `src/` folder and the public API (what a user can "use") in `include/`. The user can gather all this information without even having to look at the `README` or the documentation. This is a simple example, but it illustrates how following conventions and standards can make your project more approachable and easier to use. Projects of all sizes make use of this basic structure, such as Allegro 5, SDL, raylib, SQLite just to name a few C projects.

The specific tools and conventions will vary depending on the language and software stack of the project, but the general principles remain the same. There will always be a file (or files) that deal with dependencies, a file that deals with the build system, a folder for tests, a folder for documentation, and so on and so forth. Other times, a single file takes care of several of these tasks, such as the `pyproject.toml` file in Python, which takes care of dependencies, installation and packaging. In C projects, you will see `Make` or `autoconf` instead of `CMake`. Tools (as opposed to libraries) will typically lack a folder named `include/`, and so on and so forth.

Info

dotfiles
Dotfiles are configuration files that are used to customize the behavior of various tools and applications. They are usually hidden by default (starting with a dot). Examples you might find in a repository include, `.gitignore` (which tells git which files to ignore), `.clang-format` (which specifies the formatting rules for C or C++ code), `.travis.yml` (configuration for the CI service Travis CI), and so on. These files are usually placed in the root of the project, and are used to configure the behavior of the tools used in the project.

Advice

Specific files talk to specific tools
It is important that we follow the conventions of the language and the tools we are using. That means, for example, to choose conventional names for the different files and folders in our project. For instance, if we name the file that deals with dependencies `environment.yml`, we can navigate to the root of our project and run the command:

```
conda env create
```

`conda` will automatically search for this file and create an environment with the dependencies specified in it. Additionally, the VSCode editor will pick it up and automatically enable the environment when we open the project. Say we name this file `deps.yaml` instead. We would need to:

- Write about the file in the `README.md` so that users know how to create the environment.
- Inform conda about the file name by running `conda env create -f deps.yaml`.
- Inform VSCode about the file name by creating a `.vscode/settings.json` file.

That increases the cognitive load of the user, who now needs more information specific to your project in order to proceed.

8 Homework

8.1 Set it up

Goal

Make sure that you have a working C development environment by completing this checklist:

- ☐ Install conda (Miniforge) or Anaconda (if you are on Windows and do not want to use WSL)
- ☐ Install VSCode
- ☐ Clone/download the C template project from here
- ☐ Create a new conda environment from the `environment.yml` file in the template project
- ☐ Open the project in VSCode
- ☐ Build the project with CMake
- ☐ Run the generated tests: `cd build && ctest`

hint

- You can use `git clone https://github.com/RaulPPelaez/c-template-project.git` to clone the project. If you do not have git installed, you can install it with `conda install git`. This is one of the few times installing something in the base environment is acceptable.
- You can use the video guide here if you get stuck installing conda or VSCode.
- There will most definitely be issues along the way, such as commands requiring extra options in your system, or installers failing.
- Building with CMake consists of running the following commands from the root of the project:

```
cmake -B build
cmake --build build
```

8.2 Your first project

Goal

Reproduce the project describe in section Basic CMake syntax.

- ☐ Create a new folder for your project
- ☐ Create an `environment.yml` file with the necessary dependencies (at least `cmake`, `make`, and `c-compiler`), you can copy the one from the template project.
- ☐ Create the conda environment: `conda env create -n my_first_c_project`
- ☐ Activate the environment: `conda activate my_first_c_project`
- ☐ Open the project in VSCode: `code .`
- ☐ Create the source files: `library.c`, `library.h`, `main.c`
- ☐ Create a `CMakeLists.txt` file that describes the project
- ☐ Make sure you can build the project with CMake
- ☐ Run the generated executable

hint

- In order to build the project, you will need to run the following commands from the root of your project:

```
cmake -B build
cmake --build build
```

- The generated executable will be inside `build/`

Milestone

Move the source files to a `src/` folder. Update the top level `CMakeLists.txt` file accordingly, place the rules to build the executable and library in a new `CMakeLists.txt` file inside `src/`.

hint

- You will need to use the `add_subdirectory(src)` command in the top level `CMakeLists.txt` file.