Introduction to Programming

Lecture 4 — Developpers' tools

Nicolas Gry

Université de Saint-Étienne | Year 2026/2027

digicrea digital creativity
Université Jean Monnet Saint-Étienne

-1 Recap’ time

00 Assignment review

01 The terminal

A handy program: the *shell*

The *terminal* runs a program called the **shell**. It is an *interpreter*; it executes commands you give to it.

A handy program: the *shell*

The *terminal* runs a program called the **shell**. It is an *interpreter*; it executes commands you give to it.
You can run programs from the shell.

A handy program: the *shell*

The *terminal* runs a program called the **shell**. It is an *interpreter*; it executes commands you give to it.
You can run programs from the shell. For example, try to type:

```
code
```

Notion of *path*

Some commands let you browse through your folders and files, just like File Explorer / Finder would.

Notion of *path*

Some commands let you browse through your folders and files, just like File Explorer / Finder would.

A **path** is a location on your computer. It is made up of folders and/or files names, seperated by "/" — on Windows, they use "\", but "/" works in the shell.

You can check where your current shell is open at, by running `pwd`.

Navigating with the shell

You can use the command `cd <path>` to go the provided path. The command `ls` will show you the content of the folder you are currently in.

Navigating with the shell

You can use the command `cd <path>` to go the provided path. The command `ls` will show you the content of the folder you are currently in.
Try:

```
cd ~
pwd
ls
```

What does the terminal output?

Navigating with the shell

You can use the command `cd <path>` to go the provided path. The command `ls` will show you the content of the folder you are currently in.
Try:

```
cd ~
pwd
ls
```

What does the terminal output?
Note: ~ is called the home directory. Running `cd` with no argument will bring you there.

The root folder

Everything on your computer is a file or a folder containing one or several file(s).

The root folder

Everything on your computer is a file or a folder containing one or several file(s). There is a folder on your computer that contains all other folders and files: it is called the **root** of your computer. It's located at `/`.

Absolute path vs relative path

The **absolute path** of a file is the path that goes from root all the way to your file. It's like its full address: you can find a file from anywhere given its absolute path, just like you would find your home on maps given your full address.

Absolute path vs relative path

The **absolute path** of a file is the path that goes from root all the way to your file. It's like its full address: you can find a file from anywhere given its absolute path, just like you would find your home on maps given your full address.

A **relative path** of a file is a path that works from a given location. It's like indicating a direction from a point in a given street to the supermarket; change the origin point and the indications may become wrong.

Parent folder

`..` denotes the parent folder. For example, if your shell is open in `/Users/yourusername/Documents`, running `cd ..` will bring you to `/Users/yourusername`.

`/Users/yourusername/Documents/..` and `/Users/yourusername` denote the same location.

02 Files, folders and vscode

Organizing your coding projects 1/2

Goodbye Godbolt

Organizing your coding projects 2/2

- C++ code is written in files of extension `.cpp`.
- You should try to organize your projects using **folders**.
- You can open a folder with **Visual Studio Code** (aka *VSCode*).

03 Compiling: from a source file to a program

Your source file is not a program

The file that contains your code, the **source file**, is NOT a program. You cannot execute it.
Your code needs to be **compiled** into machine code in order to become a program. That is the job of the **compiler**.

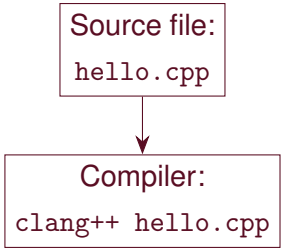
Compiling a program

Source file:

`hello.cpp`

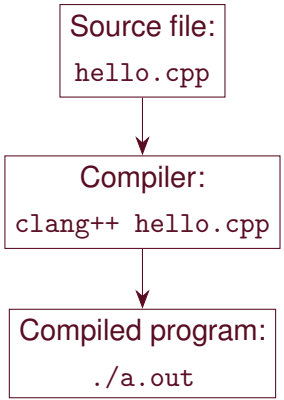
- You code is in a **source file**

Compiling a program



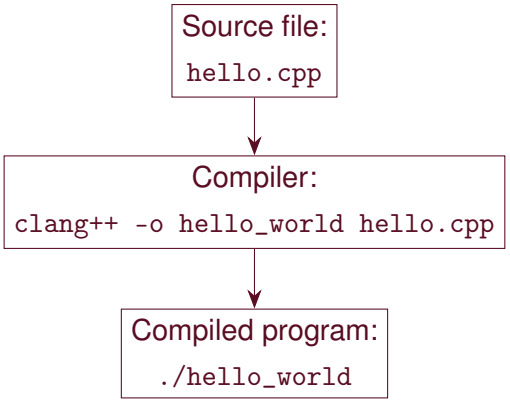
- You code is in a **source file**
- You pass that source file to a program called **compiler**

Compiling a program



- You code is in a **source file**
- You pass that source file to a program called **compiler**
- The compiler creates a **program** from your *source code*

Compiling a program



- You code is in a **source file**
- You pass that source file to a program called **compiler**
- The compiler creates a **program** from your *source code*
- You may choose the name of your program by adding the following arguments to your call to clang:

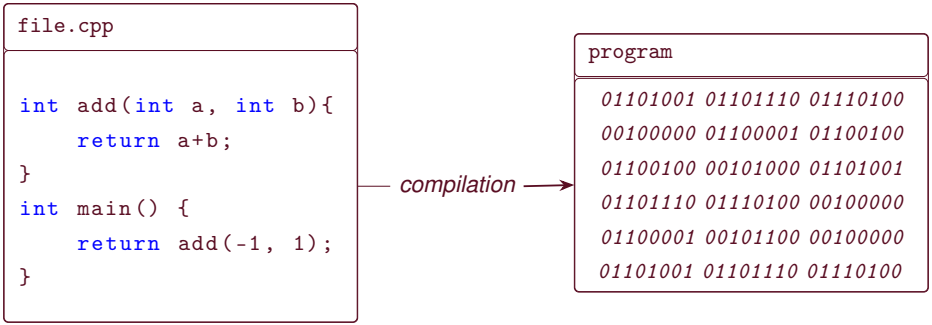
  `-o <program_name>`

From human-readable code to binary

The compiler transforms human-readable code into machine readable code. Your program contains binary code.

From human-readable code to binary

The compiler transforms human-readable code into machine readable code. Your program contains binary code.



04 Divide to conquer: multiple files for one program

The what and why of divide to conquer

In programming, “*Divide to conquer*” is a rule applied on many levels.

The what and why of divide to conquer

In programming, “*Divide to conquer*” is a rule applied on many levels.
In the case of project organization, it encourages using one file per purpose / functionality, so as to find more easily the code concerning a specific portion of a program.

The what and why of divide to conquer

In programming, “*Divide to conquer*” is a rule applied on many levels.
In the case of project organization, it encourages using one file per purpose / functionality, so as to find more easily the code concerning a specific portion of a program.
The rule *Divide to conquer* helps to avoid having files with more than a thousand lines of code, which become quickly hard to maintain.

Declaring vs Defining

Forward declaration of `display_message` in file `main.cpp`:

main.cpp

```cpp
#include <iostream>
#include <string>
void display_message(const std::string msg,
    bool breakline = true);

int main() {
    display_message("hello, world!");
    return 0;
}
```

Declaring vs Defining

Forward declaration of `display_message` in file `main.cpp`:

main.cpp

```cpp
#include <iostream>
#include <string>
void display_message(const std::string msg,
    bool breakline = true);

int main() {
    display_message("hello, world!");
    return 0;
}
```

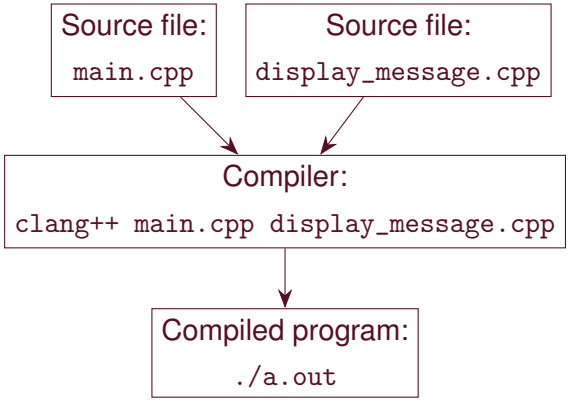
Definition of `display_message` in file `display_message.cpp`:

display_message.cpp

```cpp
#include <iostream>
#include <string>

void display_message(const std::string msg,
    bool breakline = true) {
    std::cout << msg;
    if (breakline) std::cout << std::endl;
}
```

Compiling with several files



Header files 1/2

Header files are a means to share declarations.
They are “included” in source files using `#include`. This will directly copy their content in the source file. For this reason, we put `#pragma once` at the beginning of the header file, to avoid multiple copies of the same file.

Header files 2/2

Listing 1: instruments.hpp

```
#pragma once
enum class Instrument {Piano, Guitar, Bass, Drums
    };
```

Listing 2: print.hpp

```
#pragma once
#include "instruments.hpp"
void print_instrument(const Instrument instrument)
    ;
```

Header files 2/2

Listing 4: instruments.hpp

```
#pragma once
enum class Instrument {Piano, Guitar, Bass, Drums
    };
```

Listing 5: print.hpp

```
#pragma once
#include "instruments.hpp"
void print_instrument(const Instrument instrument)
    ;
```

Listing 6: source1.cpp

```
#include "instrument.hpp"
// defines Instrument once
```

Header files 2/2

Listing 7: instruments.hpp

```
#pragma once
enum class Instrument {Piano, Guitar, Bass, Drums
    };
```

Listing 8: print.hpp

```
#pragma once
#include "instruments.hpp"
void print_instrument(const Instrument instrument)
    ;
```

Listing 9: source1.cpp

```
#include "instrument.hpp"
// defines Instrument once
#include "print.hpp"
// #pragma once encountered, won't copy instrument.hpp again
```

Compiling with header files

Header files are not source files. Since their content are copied into the source files that include them, they don’t need to be passed as argument to the compiler.

Organizing headers

Projects often have two folders:

```
./src
./include
```

Organizing headers

Projects often have two folders:

`./src`

`./include`

`src` contains the `.cpp` files. `include` contains the `.hpp` files.

Organizing headers

Projects often have two folders:

`./src`

`./include`

`src` contains the `.cpp` files. `include` contains the `.hpp` files.

You can specify the location of header files to the compiler with `-I<location>`. This way, you don't have to write the full path of your header file in the `#include` directive.

```
clang++ src/file1.cpp src/file2.cpp ... -I include -o myprogram
```

Practise Time!

05 Libraries: producing reusable code

Problematic: reusing code

There are many cases where you want to:

- Reuse code with no main function
- Reuse code without having to recompile it in every project
- Share functionalities without sharing the code behind it (usually what corporates want)

Solution: libraries

Libraries are pieces of compiled code, reusable in any project.

Solution: libraries

Libraries are pieces of compiled code, reusable in any project.

They're composed of:

- Header files, that contain the **interface**;

Solution: libraries

Libraries are pieces of compiled code, reusable in any project.

They're composed of:

- Header files, that contain the **interface**;
- `.so` (*Linux*), `.dylib` (*MacOS*), `.dll` and `.lib` (*Windows*), files which are called **dynamic libraries**

Solution: libraries

Libraries are pieces of compiled code, reusable in any project.

They're composed of:

- Header files, that contain the **interface**;
- `.so` (*Linux*), `.dylib` (*MacOS*), `.dll` and `.lib` (*Windows*), files which are called **dynamic libraries**
- `.a` files which are called **static libraries**.

Solution: libraries

Libraries are pieces of compiled code, reusable in any project.

They're composed of:

- Header files, that contain the **interface**;
- `.so` (*Linux*), `.dylib` (*MacOS*), `.dll` and `.lib` (*Windows*), files which are called **dynamic libraries**
- `.a` files which are called **static libraries**.

A library is not a program; it cannot contain a `main` function.

The intuition of libraries

The idea behind libraries is that you care more about **what** they do rather than *how* they do it.

The intuition of libraries

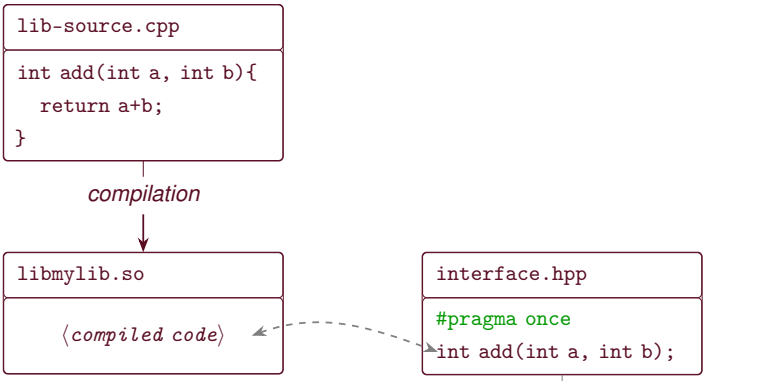
The idea behind libraries is that you care more about **what** they do rather than *how* they do it.
You get code that does something you need; you only need to know how to use it, not how it works.
Hence, you need an interface — the header files. The implementation is hidden in the compiled code — the library file(s).

What's in a library

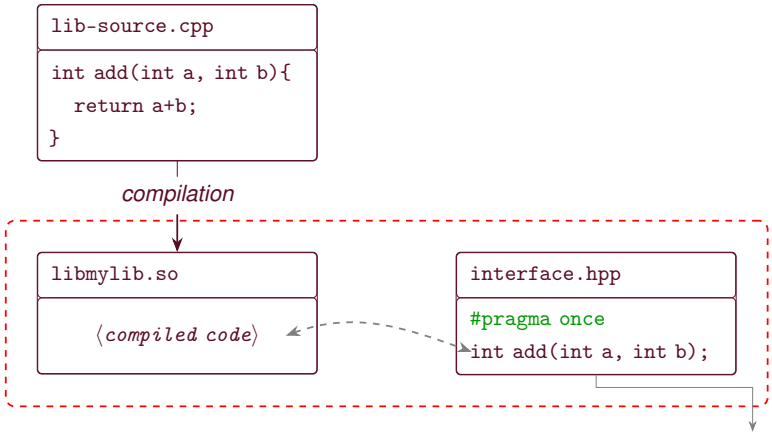
lib-source.cpp

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

What's in a library



What's in a library



How to use libraries

In code, it consists in using the libraries header files, like before:

```
#include "mylib_header.hpp"
```

How to use libraries

In code, it consists in using the libraries header files, like before:

```
#include "mylib_header.hpp"
```

To compile a program that *depends* on `libmylib.so` or `mylib.dylib`, or `mylib.dll` and `mylib.lib`, you use:

- `-lmylib` to specify the name of the library
- `-L<path to mylib file>` to specify the location of the library

Example compiling with a library

If your project is:

```
src/
└── source.cpp
include/
└── mylib_header.hpp
lib/
└── libmylib.so
```

Example compiling with a library

If your project is:

```
src/
└── source.cpp
include/
└── mylib_header.hpp
lib/
└── libmylib.so
```

Compile it with: `clang++ src/source.cpp -I include -L lib -lmylib`

Example compiling with a library

If your project is:

```
src/
└── source.cpp
include/
└── mylib_header.hpp
lib/
└── libmylib.so
```

Compile it with: `clang++ src/source.cpp -I include -L lib -lmylib`

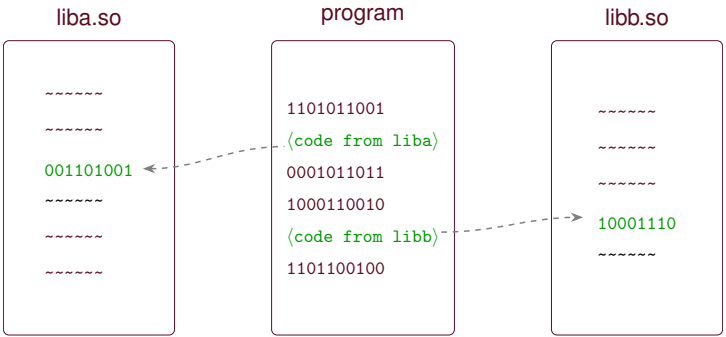
Note: `-l` options always come last in the compilation command line.

Executing a program compiled with a library 1/3

When a program is compiled with a library, the binary contained in the library *is not* embedded in the program. Rather, the program remembers what it is supposed to find in libraries and what it can find in its own binary. The library is **linked** to the program.

Executing a program compiled with a library 1/3

When a program is compiled with a library, the binary contained in the library *is not* embedded in the program. Rather, the program remembers what it is supposed to find in libraries and what it can find in its own binary. The library is **linked** to the program.



Executing a program compiled with a library 2/3

When a program is executed, it will look for its needed dependencies in regular paths like `/lib`, `/usr/lib`, `/usr/local/lib`, etc.

Executing a program compiled with a library 2/3

When a program is executed, it will look for its needed dependencies in regular paths like `/lib`, `/usr/lib`, `/usr/local/lib`, etc.
If the needed library is not in one of those location, you can add a path for your program to look into by setting an *environment variable* in the shell, before executing the program. The variable name varies depending on the OS you're on.

Executing a program compiled with a library 3/3

Windows	MacOS	Linux
`PATH`	`DYLD_LIBRARY_PATH`	`LD_LIBRARY_PATH`

MacOS/Linux

```
export [DY]LD_LIBRARY_PATH=/
    path/to/lib
./my_program
```

Windows

```
$env:PATH = "/path/to/lib;$env
    :PATH"
./my_program.exe
```

You can put several paths in this variable by seperating them with a colon `:` on MacOS and Linux, and a semicolon `;` on Windows.

Practise Time!

06 Compiling large projects with CMake

Problematic: managing large projects

Imagine you have a large project, with dozens of source files, and ten different libraries to link to your program, all located at different spots.

Problematic: managing large projects

Imagine you have a large project, with dozens of source files, and ten different libraries to link to your program, all located at different spots.

Do you really have to remember the compilation commmand, and type it everytime to compile your program?

CMake to the rescue

Fortunately not. Programmers have created **build systems** that make compiling your program much easier.
CMake is one of them.

Build system principle

Build systems have their own language.
The idea is to tell them what are your source files and dependencies — i.e libraries. This is usually done in a configuratio file, using the build system language.
Then, you can *build* your program, using the same simple command everytime.

CMake style

In the case of CMake, the configuration file is named `CMakeLists.txt`.

CMake style

In the case of CMake, the configuration file is named `CMakeLists.txt`.

First, you generate your project by running:

```
cmake -B build .
```

This *generates* your project; that is, it generates in a folder named `build` everything needed to compile your program.

CMake style

In the case of CMake, the configuration file is named `CMakeLists.txt`.

First, you generate your project by running:

```
cmake -B build .
```

This *generates* your project; that is, it generates in a folder named `build` everything needed to compile your program.

Then, you actually build your project with:

```
cmake --build build
```

What should you know?

Learning how to write a complete `CMakeLists.txt` is outside the scope of this class.

What should you know?

Learning how to write a complete `CMakeLists.txt` is outside the scope of this class. Here's what you should know or be able to do:

- When a project contains a `CMakeLists.txt`, you should have the reflex to run `cmake -B build .` followed by `cmake --build build`
- For simple projects, be able to use a reference `CMakeLists.txt`, and maybe modify it if needed

Practise time!

See you next time!