Advanced Programming for Scientific Computing (PACS)
Lecture title: Introduction

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

General Information

Lecturer: Prof. Luca Formaggia (luca.formaggia@polimi.it)
Assistant: Dr. Alberto Artoni (alberto.artoni@polimi.it)
Tutor: Dr. Beatrice Crippa (beatrice.crippa.@polimi.it)
Reception Hours: Wednesday 14.15 – 16.15 (on appointment!)

Lectures are held on
Wednesday from 10.15 to 12.00 room 3.1.3 and Thursday from 15.15 to 18.00 in room 7.0.1

The laboratory sessions are held on Friday 13.15 to 16.00 in room 3.1.4 (bring your PC with you!).

Lectures and lab sessions are recorded. Recordings will be available to registered students. Lectures and lab sessions are in presence. Remote participation is allowed only for special situations.

General Information

The course consists of lectures, laboratory sessions (you need your PC) and a project valid for the final evaluation (max 2 students, 3 for the 8 credit course)

Students following the 8 credit version of the course may replace the project with a standard written exam on request.

Also students of the 10 credit version may opt for the written exam but they will have the mark capped at 25.

Some projects may be tailored only for the 8 credits version. Here you find a list of past projects, with access to the report.

During the course, we will give assignments (challenges) using the WeBeep site of the course, consisting of questions or small exercises.
Assignments are evaluated, and will make up to 3 points at the exam.

Books and Material

C++ Primer (5th edition), S. Lippman et. al, Addison Wesley, 2012. A very good introductory book, However, not updated to the latest standards.

A tour of C++ (third edition), Bjarne Stroustrup, Pearson, 2022.

Guide to Scientific Computing in C++ J. Pitt, J. Whitely, Springer, 2012

Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers, II edition, Peter Gottschling, Addison-Wesley, 2021. Very advanced book. Lots of interesting techniques.

Modern C++ Programming Techniques for Scientific Computing. Freely available on the web. It covers also C++20.

Course Slides and Notes: on WeBeep.

On line resources

- The page of the courses on line of Politecnico, WeBeep will be used as exchange point. I will put there a copy of the slides and other material;
- A github site has been set up for the course Examples and Exercises (we will do a brief lecture on git);
- Videos on YouTube. They cover aspects related the use of git, explanation of some examples... I will try to keep them updated.
- The WeBeep page with some seminars concerning different aspects of code development.
- Lecture recordings. Through the usual channels of Politecnico di Milano (only for registered students).

On line C++ references

There are quite al lot of in-line references about C++. The main ones (in my opinion) are:

- www.cppreference.com: a very complete reference site. It is my preferred one!. A bit too technical sometimes, but you find everything!
- www.cplusplus.com: another excellent *on-line reference* on C++ with many examples, adjourned to the new standards.
- hacking c++ is another site plenty of material.
- Wikipedia is also a useful source of information.
- ChatGPT V4 can sometimes give good advice (but sometimes it makes mistakes).

Use the web to find answers!

Classroom material and engagement

In the WeBeep site, under `Material` you find

- Lectures. The slides shown at lecture.
- NotesAndArticles with set of notes and tutorials in pdf format. In particular, an introduction to the bash unix shell and on Makefiles. Some other (more techncial) notes are present, I will mention them during lecture at the right time.
- A collection of notes and hints contains... a collection of short notes and hints.... In a format that can be "printed" as a book.
- and a lot more... have a look by yourself!

The Forum, may used to post short questions/examples (also by you). The Assigment section contains the challenges.

Operative System

The reference operative system for the course is Linux, the porting of Unix to Intel-based architecture, originally developed, just for fun, by Linus Torvalds. Nowadays, Linux is the OS of choice of most servers and supercomputers, and the Linux kernel is at the base of MacOS and Android. You can either

- Use the Windows Subsystem for Linux (only for Windows systems).
- Install Linux in a virtual machine, like Virtualbox;
- Use docker (MAC with ARM processors).
- Install Linux in another partition;

My distribution of choice is Ubuntu, but you can choose another one. On WeBeep you have an introduction to the bash shell, the default command shell on Linux.

Modules

A good and recent Linux distribution is fine. However, to favour uniformity, you may use the `mk` modules.

We will give more information during the Lab sessions. However, their use is not compulsory, but it can help the one of you who do not want to play with the installation of the different packages and libraries that make up our workflow.

Examples and Lab material

They are kept as git repository on GitHub. To get them, you have first to open an account on GitHub, and store your ssh keys, as explained in this video.

Then, you open a terminal and do (<name> is a name of your choice):

```
mkdir <name> # create a directory
cd <name>
git clone --recurse-submodules  git@github.com:pacs-course/pacs-examples.git
```

To keep the content updated (do it frequently!).

```
cd <name>
git pull --recurse-submodules
```

You can run git pull from any folder of the repository.

You may watch this short video with indication on how to setul your GitHub account

Laboratory sessions

The material for the laboratory session is available on Github. To clone the repository on your PC, go in the folder where you want to store the Laboratory sessions and do

```
git clone git@github.com:pacs-course/pacs-Labs.git
```

To keep up-to-date with the laboratory sessions, do frequently

```
git pull
```

from any folder of the git repository

Some trouble-shootings

Some useful FAQ until you become git gurus (you will by the end of the course):

Q I have inadvertently erased file `pippo.cpp` from the Example repository in my PC. What can I do?
A `git checkout pippo.cpp`

Q I have inadvertently modified file `pippo.cpp` of the Example repository in my PC and now the example does not compile anymore. How can I recover the unmodified version?
A `git checkout pippo.cpp`

Q I have found a bug in `pippo.cpp`, I fixed it, and I want the teacher to implement my correction! What do I have to do?
A You are a good guy! You should create a pull request:

```
git commit -a -m "my nice correction"
git push
```

The main folders of the Examples

The example shown during the course are organised in different directories

- `Extra` Some extra material: software, notes etc.
- `Examples` The examples.

In (almost) all directories a `README.md` files contains the description of the content.
In the `Example` folder you have also the file `DESCRIPTION.md` with an overall description.

Read the README, the name has been chosen for some reason...

A glance of the Examples folder

The directory `Examples` consists of several subdirectories:

- `include`, where the header files used by more than one examples are stored;
- `lib`, where libraries used by more than one example are stored;
- `src`, where the actual examples are stored.

Follow the instructions in the `README.md` files to be able to compile the examples.

You can also watch this video, which explains how to download the GiHub repo and what are the first steps to take.

Some notes on the external utilities

- Some utilities are provided by external sources. In particular: in the `Extras` directory you have `json`, `muparser` and `muparserx`. To install, follow the instructions in the `README.md` or, when present, the `README_PACS.md` file.
- Other utilities provided by external sources are in the folder `src/LinearAlgebra`. In particular, `redsvd_h`, `CPPNumericalSolvers` and `spectra`. Again, to compile look at the `README_PACS.md` file.
- To compile some of the external packages you need to have cmake installed.
- The external utilities are not available if you forgot the `--recurse-submodules` when cloning the repo. In this case, use the `install-git-submodules.sh` script.

Compilers

We use as reference compiler the gnu compiler (g++), at least version 9.0. It is normally provided with any Linux distribution. Check the version with `g++ -v`.

Another very good compiler is clang++, of the LLVM suite, downloadable from llvm.org. Use version 9.0 or higher. You may find it in most Linux distributions (on ubuntu you install it with `sudo apt-get install clang`). With respect to the gnu compiler it gives better error messages (and is sometimes faster).

We will stick to C++ standard, so in principle any compiler which complies to the standard should be able to compile the examples.

Integrated Development tools

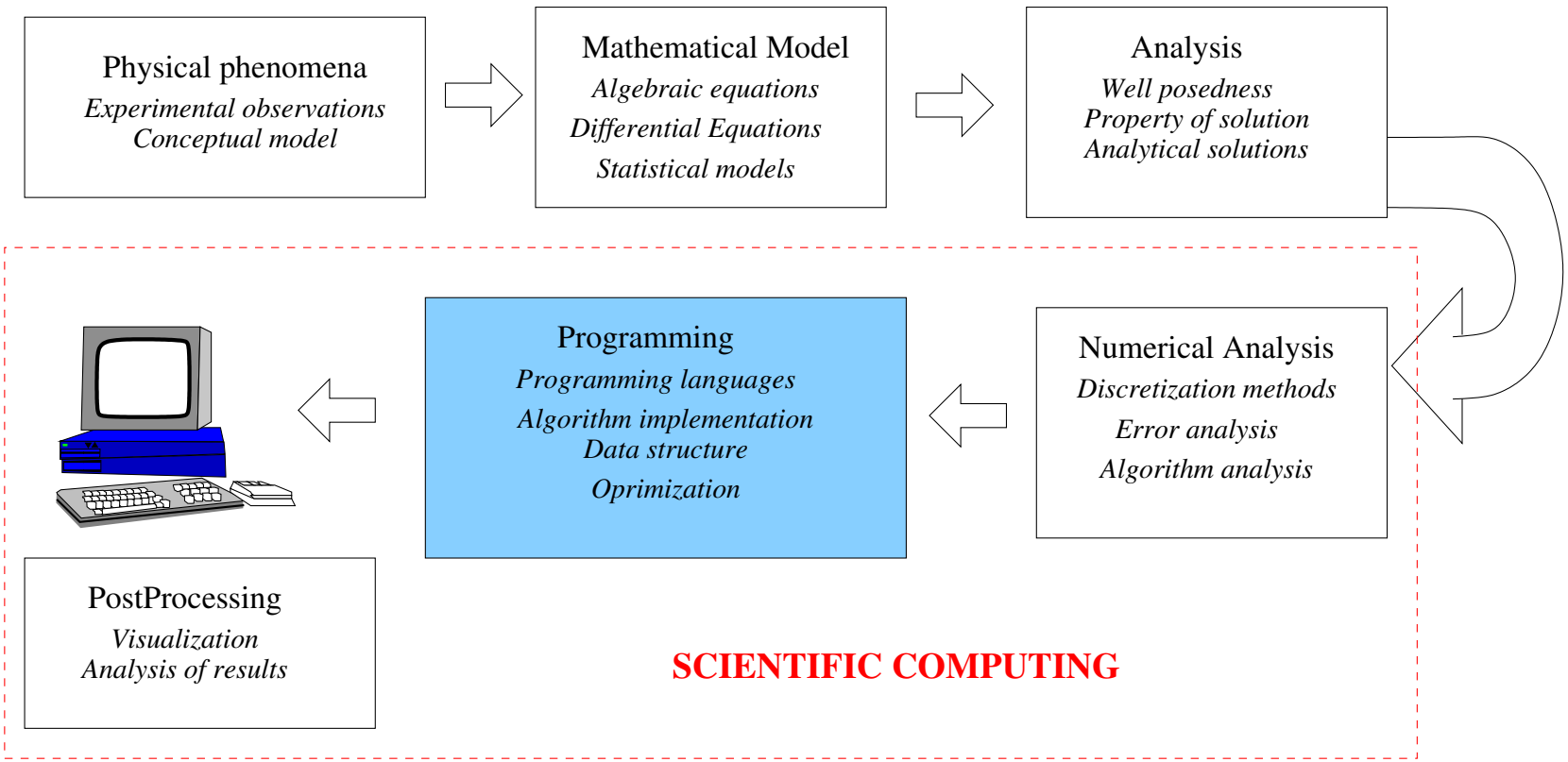
The use of IDEs (Integrated Development environment) may help the development of a software.

- Visual Studio Code An excellent IDE. Typing `CTRL P` you can access the panel to set up the C++ preferences. Set the C++ version to 20 and possible additional directories to look for header files, in particular the Eigen libraries for linear algebra, if they are not stored in a standard place.
- CLion. Very popular but strongly based on cmake.
- eclipse Now a bit outdated.

All examples illustrated in the course contain Makefile to ease compilation (I will make a lecture on Makefiles!). Add the Makefile support to the chosed IDE, if not yet available.

It is not compulsory to use an IDE (but it helps). A good editor may be sufficient. Good editors are Atom, emacs, vim and gedit. They all support syntax highlighting.

From physics to computer



Why C++

C++ is

- Reasonably efficient in terms of CPU time and memory handling, being a compiled language;
- In high demand in industry;
- A (sometimes exceedingly) complex language: if you know C++ you will learn other languages quickly;
- A strongly typed[1] language: safer code, less ambiguous semantic, more efficient memory handling;
- Supporting functional/object oriented and generic programming;
- Backward compatibile (unlike Python...). Old code compiles (almost) seamlessly.
- It is green!

[1] Not everybody agrees on the definition of *strongly typed*.

Popular languages (TIOBE 2023)

The first 5 most popular computer languages by TIOBE:

Jan 2024	Jan 2023	Change	Programming Language	Ratings
1	1		Python	13.97%
2	2		C	11.44%
3	3		C++	9.96%
4	4		Java	7.87%
5	5		C#	7.16%

C++ has been the fastest growing language in 2022, and is now stably at the 3rd place.

Alternatives for scientific computing

- Python Very effective in building up user interfaces and connect to code written in other languages. Many modules for scientific computing and statistics, as well as machine learning. In can be interfaced with C++ using pybind11.
- FortranXX. Very good set of intrinsic mathematical functions. Can produce very efficient coding for mathematical operations. Support for HPC.
- C. Much simpler than C++, it lacks the abstraction of the latter. Many commercial codes for engineering simulations (and the Linux kernel) are written in C.
- Matlab/R They are essentially interpreted languages. Octave is a good free alternative for Matlab, with a nice interface to C++.
- Rust A simple and fast compiled language with a rich type system and reliable memory handling. The use in scientific computing is on the rise, but still a niche language.

Some useful C++ libraries

Normally you don't want to reinvent the wheel, and there are many and many C++ or C++ compatible libraries that you can integrate in your code. Besides the one hosted in the course Examples repository, we mention

- Eigen for linear algebra.
- SuiteSparse for high performance linear algebra on sparse matrices.
- Stat++ and Stan Math for statistics, automatic differentiation and Baysian inference.
- tensorflow, mlpack and OpenNN for machine learning and neural networks.
- The boost libraries. A huge set of C++ libraries.

Many libraries are directly provided in Linux distributions. An extensive list is here.

Software organization

A typical C++ program is organised in header and source files. header files contain declarations for functions, classes, variables, and macros that you want to use in other C++ source files. C++ header files have extension `.hpp` and are eventually stored in special directories with name `include`.

Source files A C++ source files, typically with the extension `.cpp` or `.cc`, contain the actual implementation code for your program, and are normally collected under the directory `src`

Libraries can be compiled (static or shared) or template libraries. A compiled library is collection of compiled objects that can be used (linked) by an external program. It is formed by header and library files (extension `.a` or `.so`). A template library consists only of header files. We will have a special lecture on libraries

What a header file should not contain

Function definitions	double norm(){ ..}
Method definitions	double Mat::norm(){ ..}
Definition of non-constexpr variables	double bb=0.5;
Definition of static class members	double A::b;
C array definitions	int aa[3]={1,2,3};
Array definitions	std::array<double,3> a{1,2,3};
Unnamed namespaces	namespace { ...}

Which type of information a header file provides?

A header file contains all the information needed by the compiler to verify existence of types (a part plain old data types), calculate the dimension of an object of the given type, instantiate templates, define static objects (i.e. objects that are completely defined at compilation stage, and not run-time).

This information is shared by all translation units that include the header file via the **#include** directive.

Remember that the `#include` directive does exactly what it says: it includes the content of the specified file.

```
#include <vector> //<- includes the header file vector
...               //    of the standard library
```

What should a header file contain

- Declaration of Functions and Classes: Header files are used to declare functions and classes that are implemented in corresponding source (.cpp) files.
- Type Definitions: define new types such as structs, enums, and typedefs which are used across multiple source files.
- Template Definitions: Templates are dfined in header files because must be available in all source files that instantiate them.
- Inline Functions and Variables.
- Global Variables: Although their use is discouraged, they can be declared in header files using the **extern** keyword.
- Constants Expression: Constants that need to be shared across different source files.

What a header file should not contain

Function definitions	double norm(){ ..}
Method definitions	double Mat::norm(){ ..}
Variable definitions	double bb=0.5;
Definition of static class members	double A::b;
C array definitions	int aa[3]={1,2,3};
Array definitions	std::array<double,3> a{1,2,3};
Unnamed namespaces	namespace { ...}
using namespace directives	using namespace std

A note on header files in modern C++

The increasing use of templates, constexpr functions, automatic functions...., whose definitions are in header files, is making the use of source files less and less relevant (a parte the main file, of course). There are entire libraries (the Eigen library of linear algebra for instance), made only of header files!.

Yet, in several situation the distinction of header file and source file is still very relevant (for dynamic libraries for instance, or in more classical non-template programming.)

Therefore, in the following we try to clarify the compilation process assuming to have a full set of header and source files, where among the latter one is the main file.

Translation unit

A C++ translation unit (also called compilation unit) is formed by a source file and all the (recursively) `included` header files it contains.
A program is normally formed by more translation units, one and only one of which is the main program.

The important concept to be understood is that during the compilation process each translation unit is treated separately, until the last stage (linking stage).

The compilation steps (simplified)

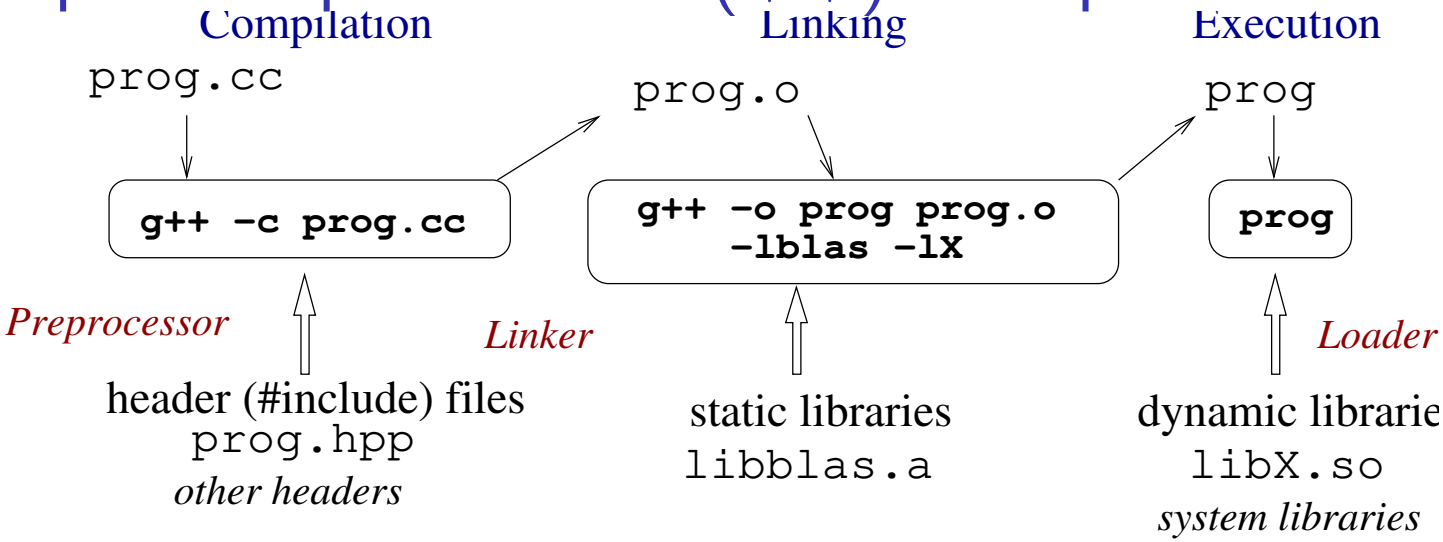
Compiling an executable is in fact a multistage process formed by different components. The main ones are

- Preprocessing. Each translation unit is transformed into an intermediate source by processing CPP directives;
- Compilation proper. Each preprocessed translation unit is converted into an object file. Most optimization is done at this stage;
- Linking Object files are assembled and unresolved symbols resolved, eventually by linking external static libraries, and an executable is produced.

When you launch the executable, you have an additional step:

- Loading Possible dynamic (shared) library are used to complete the linking process. The program is then loaded in memory for execution.

The compilation process in C(++) - simplified-



Command `g++ -std=c++20 -o prog prog.cc` would execute both *compilation* and *linking* steps.

MAIN g++ OPTIONS. Some are preprocessor or compiler or linker options. Mixed color means that they apply to more than one stage.

-g	For debugging	-O[0-3] -Ofast	Optimization level
-Wall	Activates warnings	-Idirname	Directory of header files
-DMACRO	Activate cpp MACRO	-Ldirname	Directory of libraries
-o file	executable in file	-lname	link library libname.a|so
-DMACRO=V	set cpp MACRO	-std=c++20	activates c++20 features
-fopenmp	activates openMP	-c	create only object file

The default C++ version for g++ versions 9 and 10 is c++14. We use the c++20 extension.

The compilation process

But in fact the situation is usually more complex, we normally have more than one translation units (source files)

```
g++ -std=c++20 -c a.cpp b.cpp
g++ -std=c++20 -c main.cpp
```

produce the object files `a.o`, `b.o` and `main.o`. Only one of them contains the main program (`int main()...`).
Then,

```
g++ -std=c++20 -o main main.o a.o b.o
```

produces the executable `main` (linking stage).

Each translation unit is compiled separately, even if they are in the same compiler command.

All in one go

Of course it is possible to do all in one go

```
g++ -std=c++20 main.cpp a.cpp b.cpp -o main
```

But it is normally better to keep compilation and linking stages separate. If you modify `a.cpp` you have to recompile only `a.o` and repeat the linking.

Note however, that the compiler will in any case treat the compilation units `a.cpp`, `b.cpp` and `main.cpp` separately.

The preprocessor

To understand the mechanism of the header files correctly it is necessary to introduce the C preprocessor (`cpp`). It is launched at the beginning of the compilation process and it modifies the source file producing another source file (which is normally not shown) for the actual compilation.

The operations carried out by the preprocessor are guided by directives characterized by the symbol # as the first character of the line.

Synopsis of cpp

Very rarely one calls the preprocessor explicitly, yet it may be useful to have a look at what it produces
To do that one may use the option `-E` of the compiler:
`g++ -E [-DVAR1] [-DVAR2=xx] [-Iincdir] file >pfile`

Note: `-DXX` and `-I<dirname>` compiler options are in fact **cpp options**.

The first indicates that the preprocessor macro variable `XX` is set, the second indicates a directory where the compiler may look for header files.

Main cpp directives

All preprocessor directives start with a hash (#) at the first column.

`#include<filename>`

Includes the content of `filename`. The file is searched fist in the directories possibly indicated with the option `-Idirname`, then in the system directories (like `/usr/include`).

`#include "filename"`

Like before, but first the current directory is searched for `filename`, then those indicated with the option `-Idir`, then the system directories.

`#define VAR`

Defines the *macro variable* `VAR`. For instance `#define DEBUG`. You can test if a variable is defined by `#ifdef VAR` (see later). The preprocessor option `-DVAR` is equivalent to put `#define VAR` at the beginning of the file. Yet it overrides the corresponding directive, if present.

`#define VAR=nn`

It assigns value `nn` to the (*macro variable*) `VAR`. `nn` is interpreted as an alphanumeric string. Example: `#define VAR=10`. Not only the test `#ifdef VAR` is positive, but also any occurrence of `VAR` in the following text is replaced by `10`. The corresponding cpp option is `-DVAR=10`.

```
#ifdef VAR
code block
#endif
```

If `VAR` is undefined code block is ignored. Otherwise, it is output to the preprocessed source.

```
#ifndef VAR
code block
#endif
```

If `VAR` is defined code block is ignored. Otherwise, it is output to the preprocessed source.

Special macros

The compiler set special macros depending on the options used, the programming language etc. Some of the macros are compiler dependent, other are rather standard:

- __cplusplus It is set to a value if we are compiling with a c++ compiler. In particular, it is set to 201103L if we are compiling with a C++11 compliant compiler.
- NDEBUG is a macros that the user may set it with the `-DNDEBUG` option. It is used when compiling "production" code to signal that one DOES NOT intend to debug the program. It may change the behavior of some utilities, for instance assert() is deactivated if NDEBUG is set. Also some tests in the standard library algorithms are deactivated. Therefore, you have a more efficient program.

EXAMPLES

Some examples on the way the preprocessor works are in Preprocessor.

The header guard

To avoid multiple inclusion of a header file the most common technique is to use the header guard, which consists of checking if a macro is defined and, if not, defining it!

```
#ifndef HH_MYMAT0__HH
#define HH_MYMAT0__HH
... Here the actual content
#endif
```

The variable after the `ifndef` (`HH_MYMAT0__HH` in the example) is chosen by the programmer. It should be a long name, so that it is very unlikely that the same name is used in another header file! Some IDEs generate it for you!

Testing for the C++ standard you are using

In Utilities/cxxversion.hpp an example of the use of cpp macro to test in the program the C++ standard one has used to compile a program.

The file Utilities/test_cppversion.cpp shows an example of its use. This code is useful also to remember the value that __cplusplus may assume!

The compilation proper

After the preprocessing phase the translation unit is translated into an `object code`, typically stored in a file with extension `.o`.

Object code, however, is not executable yet. The executable is produced by gathering the functionalities contained in several object files (and/or libraries).

```
g++ -std=c++20 -c -Wall a.cpp b.cpp
```

run preprocessig+compilation proper and produces the object files a.o and b.o.

The linking process

The process to create an executable from object files is done by calling the linker using *the same name of the compiler used in the compilation process*.

```
g++ main.o a.o b.o -lmylib -o myprogram
```

The linker is called with the same name of the compiler (g++ in this case) so it knows which system libraries to search! Here, it will search the c++ standard library. If you call the standalone linker, called ld you need to specify yourself where the c++ standard library resides!

You have to indicate possible other libraries used by your code. In this case the library `libmylib`.

The loader and shared (dynamic) libraries

We will make a lecture on shared libraries. For now, I just say that part of the linking process is postponed to the moment in which the program is launched. This last step is performed by the `loader`, which you never call directly, it is handled by the operative system.

To conclude this overview

Suppose your program is formed by the files `main.cpp`, `a.cpp`, `b.cpp` e gli header files `a.hpp`,`b.hpp`, stored in `../include`. And that it need to use the Eigen template library, formed y only header files in `/usr/local/include/eigen3` and you need the lapack library `liblapack.so` stored in the standard directory `/usr/lib`.

```
g++ -std=c++20 -Wall -g -c -I../include \
-I../usr/local/include/eigen3 main.cpp a.cpp b.cpp
g++ -g -o main main.o a.o b.o -llapack
```

The first line produce `main.o a.o b.o`. `-Wall` activates all warnings, `-g` because you want to activate the possibility of using the debugger (no optimizazion: it implies `-O0`). The second lines links the object files together and with the lapack library to produce the executable `main`.

A graphical view of the compilation process