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

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

General Information

Lecturer: Prof. Luca Formaggia (luca.formaggia@polimi.it)
Assistant: Dr. Matte Caldana (matteo.caldana@polimi.it)
Tutor: Dr. Paolo Joseph Baioni (paolojoseph.baiponi@polimi.it)
Reception Hours: Wednesday 14.15 – 16.15 (on appointment!)

Lectures are held on
Wednesday from 10.15 to 12.00 room 25.2.3 and Friday from 13.15 to 16.00 in room 7.0.1.

The laboratory sessions are held on Thursday from 15.15 to 18.00 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 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 little 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.

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.

The material shown during the laboratory sessions is in the Lab folder in the same git repo of the Examples.

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

- install Linux in a virtual machine, like Virtualbox (recommended choice, but you need a PC with enough RAM);
- install Linux in another partition (but you need to know what you are doing);
- install the Windows Subsystem for Linux (only for Windows systems).

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.

Versioning system: git

Git is a free and open source distributed version control system. Originally developed by Linus Torvalds (the inventor of the linux kernel) is now used for the development of hundreds of open source and commercial software projects.

By using git we can

- Keep track of all modifications and additions (it is possible to be notified by email!);
- Have a forum for discussion (but for that there is also Beep);
- Allow students to contribute (you may clone the git repo on your PC, make changes, and do a pull request).
- Git may be used also for the project for your exam!.
- For collaborative work we use GitHub, but other sremote repository are possile, like Bitbucket

It is important to have some basics on git from the start since it is used for Examples and Labs.

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 --recursive  git@github.com:pacs-course/pacs-examples.git
```

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

```
cd <name>
git pull
```

In fact. you can run git pull from any folder of the repository.

Some trouble-shootings

Some useful FAQ undil you are a git guru (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 Type `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 Type `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 boy/girl! You should create a pull request But, until you know git better, maybe (so far) it is simpler to send an email with the corrected file.

The main folders

The example shown during the course are organised in different directories

- `Labs` The exercises of the laboratory sessions.
- `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 main `Example` folder you have also the file `DESCRIPTION.md` with an overall description.

Overlook of 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.

In each directory under `src` there is a *Makefile*: typing `make` will compile the example; `make doc` will produce a documentation in the subdirectory *doc*, `make clean` will do a cleanup ; `make distclean` will cleanup also the documentation.

Submodules

Parts of the Examples are kept in other git repositories, since they are forked from software made and updated by others.

The `--recursive` in `git clone` will also download the submodules. If you want to keep them updated with the remote repo (or you have forgotten the `--recursive` when cloning the repo) do

```
git submodule update --recursive --remote --merge
```

But, in fact, I have created a script, install −git−submodules.sh, for the purpose.

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 `--recursive` when cloning the repo. In this case, use the `install-git-submodules.sh` script.

Compilers

The reference operative system for the course is Linux.
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.

Development tools

The use of IDEs (Integrated Development environment) may help the development of a software. I often use eclipse (downloadable from www.eclipse.org and provided by several Linux distributions), but it does not mean that it is the best!

Other very well known IDEs are Code::Blocks and CLion.
All examples illustrated in the course will contain a Makefile to ease compilation (I will make a lecture on Makefiles!).

Of course, 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. nano is a lightweight texteditor, available almost everywhere.

A note on the language used in the course

The course is given in English. Please forgive my mistakes, bad pronunciation and typos.
I suggest a book for those of you who wish to write the project report or the thesis in English. It contains also a lot of hints on the use of LaTeX.

N.J. Higham, Handbook of Writing for the Mathematical Sciences, Second Edition, SIAM, ISBN: 978-0-89871-420-3, 1998.

Get it, it is really plenty of good advice (and nice quotations).

A note on the author: Nicholas, J. Higham, is a well known mathematician, famous for his works on the accuracy and stability of numerical algorithms. He has contributed software to LAPACK and the NAG library, and in several pieces of code currently included in MATLAB.

Code documentation

Documenting a code is important. I use Doxygen for generating reference manuals automatically from the code. To this aim, `Doxygen` requires to write specially formatted comments. We will show examples during the course and during the exercise sessions. The web site contains an extensive manual and examples.

Beside providing "*doxygenated*" comments to introduce classes and methods, it is important to comment the source code as well, in particular the critical parts of it.

Do not spare comments, but avoid meaningless ones and ...
maintain the comments while maintaining your code:
a wrong comment is worse than no comment.

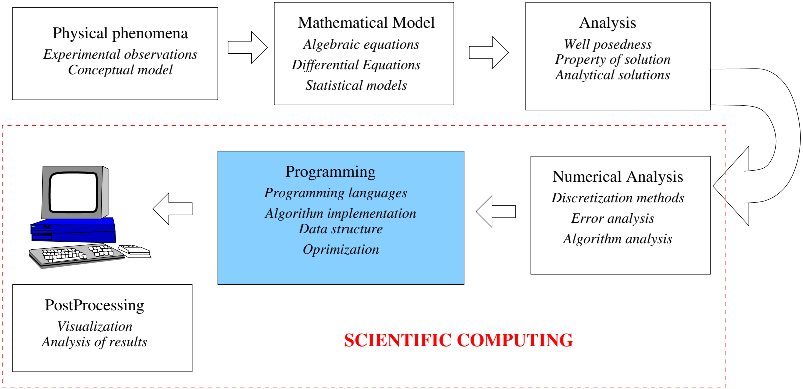
How to generate doxygen documentation of the examples

- edit the DoxyfileCommon in the Examples root directory, replacing the last line in

  ```
  INCLUDE_PATH = ./include \
                 . \
  /home/forma/Work/pacs/PACSCourse/Material/Examples/incl
  ```

  with MyRootDirectory/include (full path!).
- Copy the file DoxyfileCommon in the folder where you want to generate documentation, renaming it Doxyfile
- run doxygen
- in ./doc/html you find the documentation in html, just run your favourite browser and load index.html.

From physics to computer



Physical phenomena

Experimental observations
Conceptual model

Mathematical Model

Algebraic equations
Differential Equations
Statistical models

Analysis

Well posedness
Property of solution
Analytical solutions

Numerical Analysis

Discretization methods
Error analysis
Algorithm analysis

Programming

Programming languages
Algorithm implementation
Data structure
Oprimization

PostProcessing

Visualization
Analysis of results

SCIENTIFIC COMPUTING

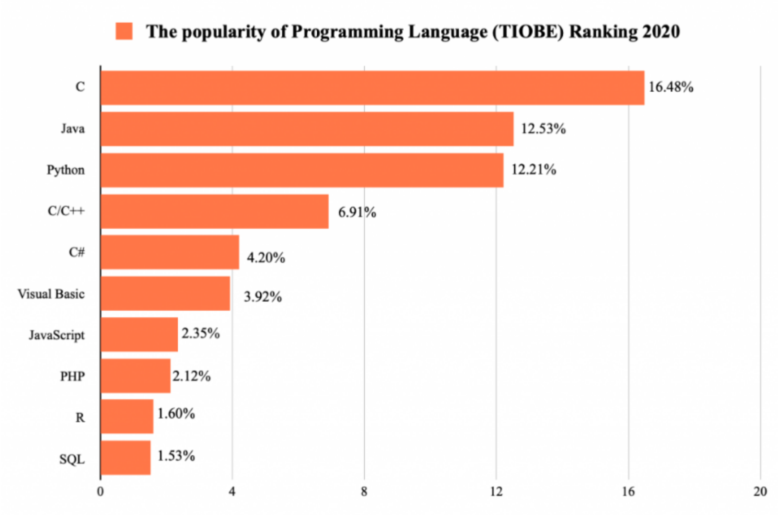
Why C++

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)



C++ has been the fastest growing language in 2022 !Read the news here.

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 (andthe Linux kernel) are written in C.
- Matlab/R They are essentially interpreted languages. Octave is a good free alternative for Matlab, with 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 one doesnt want to reinvent the well, 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 perfomance linear algebra on sparse matrices.
- Stat++ and San Math for statistics, automatic differentiation and Baysian inference.
- tensorflow, mlpack and OpenNN for machine learning and neural networks.

Many other libraries are available and we will use some of them during the course. Many are available directly in Linux distributions. An extensive list is here.

Some nice utilities: GetPot and Json++

To be able to input parameters and simple data from files, or from the command line, we provide two utilities in the course repository: GetPot and JSON for Modern C++ The former is in src/Utilities, the second is a submodule in Extras/json (look at the `README_PACS.md` file to install it for the other examples).

For simple visualization of results, the program gnuplot is a valid tool. It allows to plot the results on the screen or produce graphic files in a huge variety of formats. It is driven by commands that can be given interactively or through a script file.

An interesting add-on that allows `gnuplot` to be called from within a program is gnuplot-iostream, and is provided in src/Utilities.

The Eigen library

The Eigen library (Version 3) is a library of high performance matrices and vectors that we will use often during the course. I will give the details in another lecture. Yet, I am mentioned it here since you need to have it installed to run some of the examples.

If you use the module architecture that will be explained during exercise session you have nothing to do, Eigen library is available. If not you can install it from a package for your distribution or simply download it from the web and follow the instruction (quite simple, is a template only library, no compilation needed to install it!).

If not provided with the modules, you should modify the `Makefile.inc` file to specify where the directory with the header files are contained.

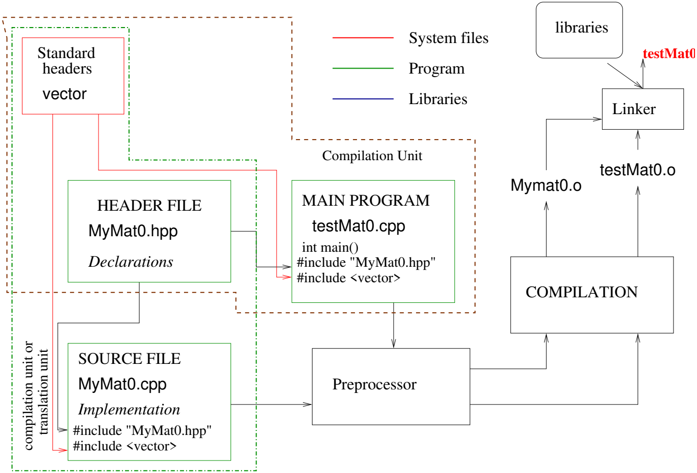
Software organization

A typical C++ program is organised in header and source files. header files contains the information that describes the public interface of your code: declarations, definition of function class templates, etc. C++ header files have extension `.hpp` and are eventually stored in special directories with name `include`.

Source files contain the implementations (definitions of variables/functions/methods) and are normally collected under the directory `src`

Libraries can be static or dynamic (shared), or template libraries (we will have a special lecture on libraries). For the moment let's think the library as a collection of compiled object that can be used (linked) by an external program. A *template library* consists only of header files, so it goes in `include/`.

A possible layout



What should a header file contain

Named namespaces	namespace LinearAlgebra
Type declarations	class Matrix{...}
Extern variables declatations	extern double a;
Constant variables	const double pi=4*atan(1.0)
Constant expressions	constexpr double h=2.1
Constexpr functions	constexpr double fun(double x){...}
Enumerations	enum bctype {...}
Function declarations	double norm(...);
Forward declarations	class Matrix;
Includes	#include<iostream>
Preprocessor directives	#ifdef AAA
Template declarations	template <class T> class A;
Template definitions	template <class T> class A{...};
Inline functions	inline fun(){...}
Inline variables	inline double x;
typedefs	typedef double Real;
type alias	using Real=double;

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 at linking stage).

This information is shared by all translation units that uses the tools declared in the header file. This is accomplished 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
```

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)

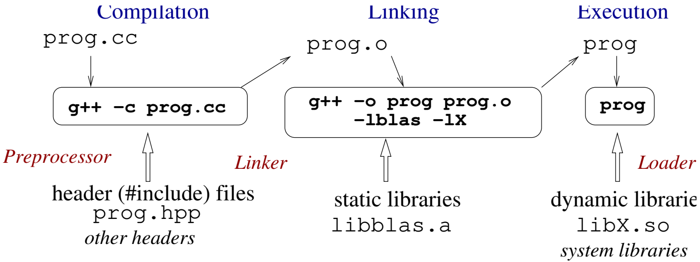
Compiling an executable is in fact a multistage process formed by different components. The main ones are

- 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-



Compilation

prog.cc

g++ -c prog.cc

Preprocessor

header (#include) files
prog.hpp
other headers

Linking

prog.o

g++ -o prog prog.o
-lblas -lX

Linker

static libraries
libblas.a

Execution

prog

prog

Loader

dynamic librarie
libX.so
system libraries

Command `g++ -std=c++17 -o prog prog.cc` would execute both *compilation* and *linking* steps.

MAIN g++ OPTIONS (some are in fact preprocessor or linker options)

-g	For debugging	-O[0-3] -Ofast	Optimization level
-Wall	Activates warnings	-Idirname	Directory of header files
-DMACRO	Activate MACRO	-Ldirname	Directory of libraries
-o file	output in file	-lname	link library
-std=c++14	activates c++14 features	-std=c++20	activates c++20 features
-std=c++17	activates c++17 features	-c	create only object file

The default C++ version for g++ versions 9 and 10 is c++14. In version 11 is c++17.

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++17 -c a.cpp b.cpp
g++ -std=c++17 -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++17 -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++17 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.

Let's look at the 3 stages in more detail.

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 # in the first column. The operations are rather general, and the C preprocessor may be used non only for C or C++ programs but also for other languages like FORTRAN!.

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`.

An example of command parsing with GetPot

```
int main (int argc, char** argv)
GetPot    cl(argc, argv);
// Search if we are giving -h or --help option
if( cl.search(2, "-h", "--help") ) printHelp();
// Search if we are giving -v version
bool verbose=cl.search("-v");
// Get file with parameter values
// with option -p filename
string filename = cl.follow("parameters.pot","-p");
```

Get what follows -p. But if -p is not present use the first argument as default filename

Usage:
-p = name
-p name
if an option after -p is required

This enable to parse options passes on the command line: if you rune

```
main -p file.dat
```

filename in the code will contain the string `file.dat`.

An example of GetPot file

GetPot allows also reading parameters from a file:

```
# You can insert comments everywhere in the file.
#
a=10.0  # Assigning a number.
vel=45.6
index=7
[parameters] # A section.
sigma=9.0
mu=7.0
[./other]  # A subsection.
p=56
```

Reading from a getpot file

```
std::ifstream file(``parameters.dat'');
GetPot gp(file);// read the file in a getpot object
double a=gp(``a'',0.0);// 0.0 is the default if a not there
// If you use automatic deduction of type, the type is that
// of the defualt value. In this case, float
auto sigma=gp(``parameters/sigma'',10.0);
auto p=gp(``parameters/other/p'',10.0);
   ...
```

Many other tools and goodies may be found in the online manual at GetPot.

Examples of Json++: a json file

```
{
    "answer": {
        "everything": 42
    },
    "happy": true,
    "list": [1,0,2],
    "stringlist":["first","second","third"],
    "name": "Niels",
    "nothing": null,
    "object": {
         "currency": "USD",
          "value": 42.99
            },
     "pi": 3.141
 }
```

Unfortunately json files do not allow for comments (a real pity).
But json format is a standard.

Reading the json file using the provided utility

See the code in Extras/json/MyExamples/test.cpp

Another nice utility: gnuplot

Gnuplot, see www.gnuplot.info, is a portable command-line driven graphing utility originally created to allow scientists and students to visualize mathematical functions and data interactively.

It is simple and portable to various architecture. We will see example of its use. More details on the indicated web site.

With gnuplot-iostream, provided in `stc/Utilities`, you can even create the plot of the solution from within the code. See instructions in the indicated web site (it is used in some examples).