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.

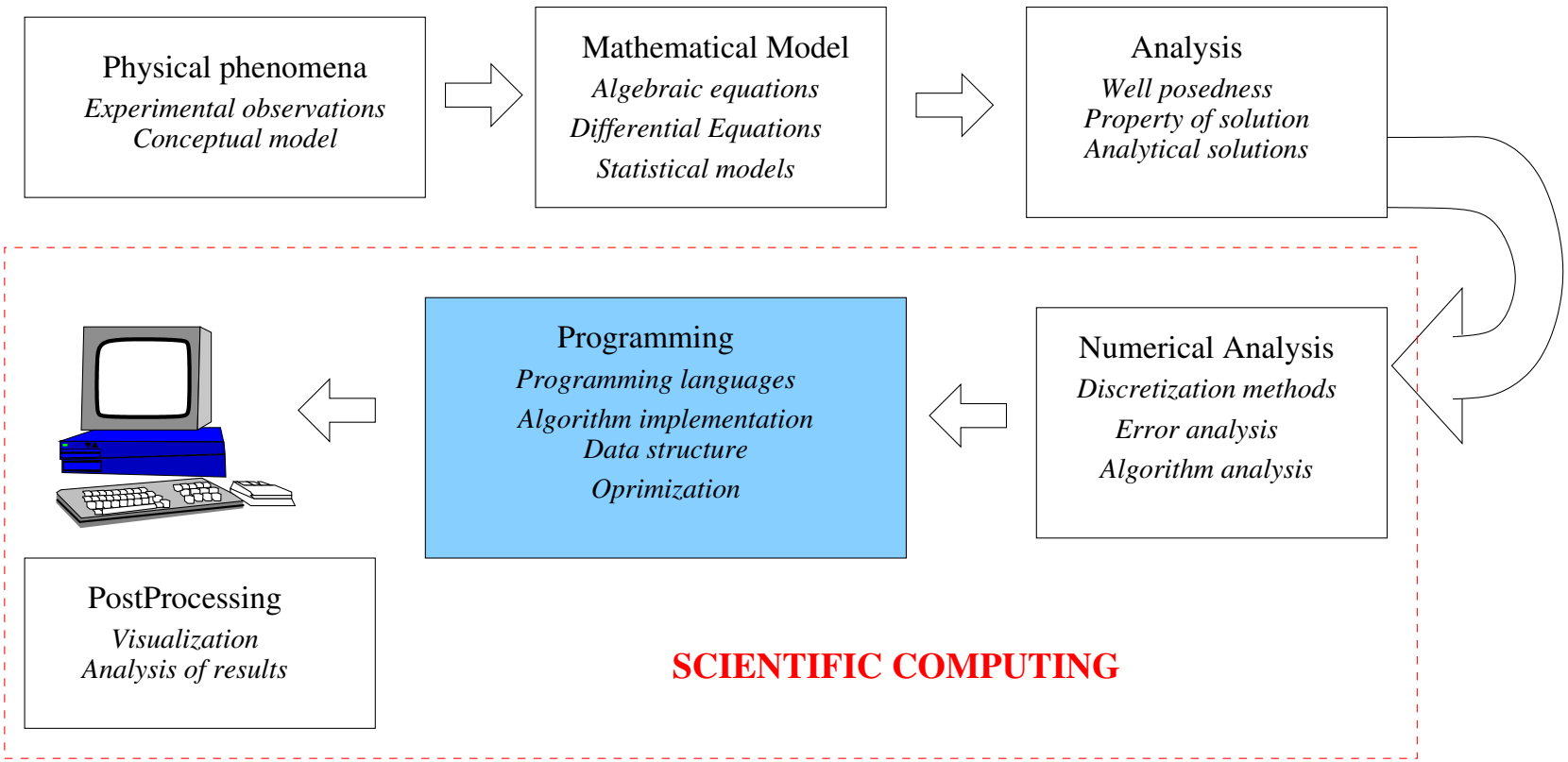
How to generate doxygen documentation of the examples

- 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



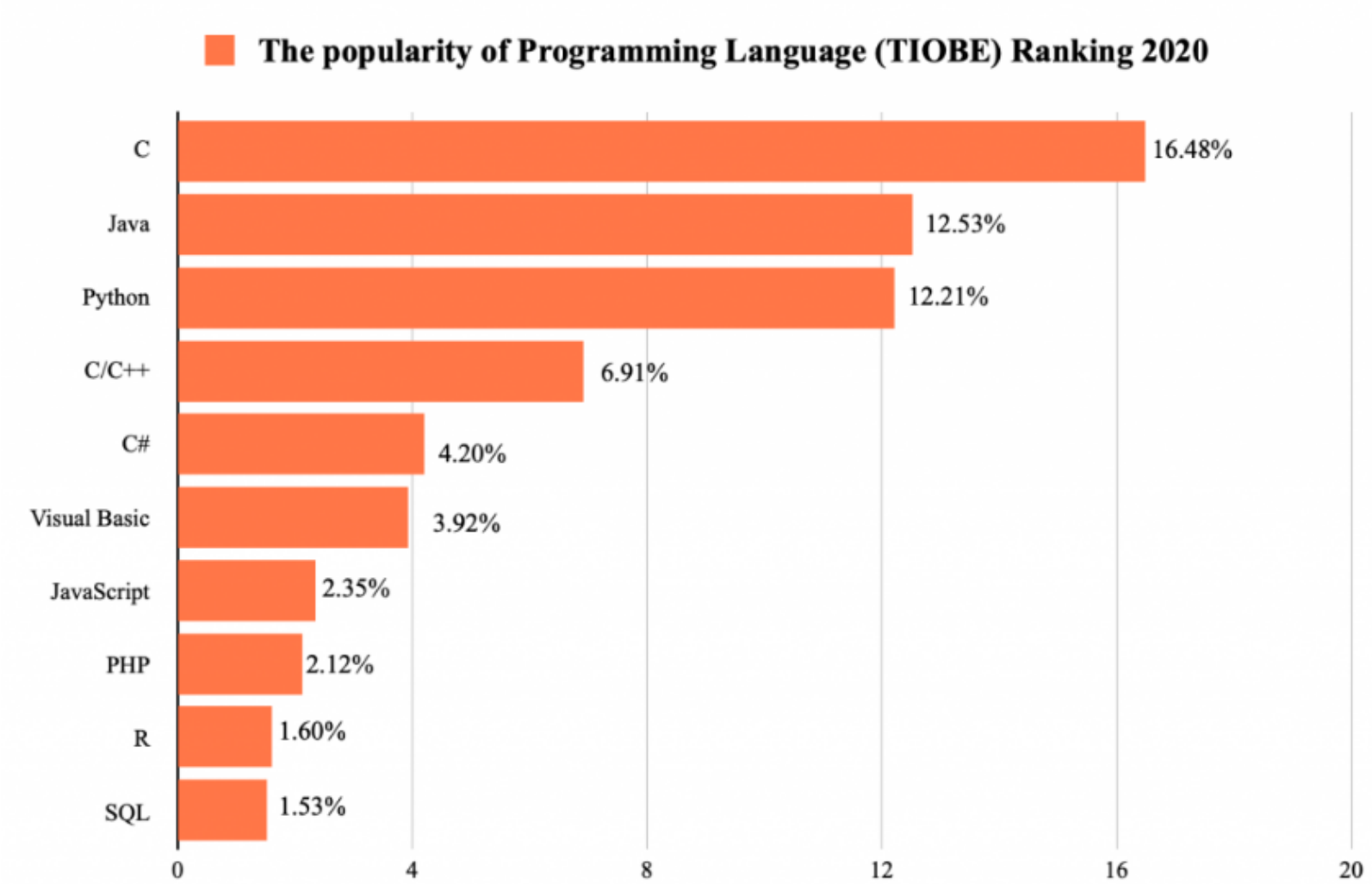
Why C++

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.

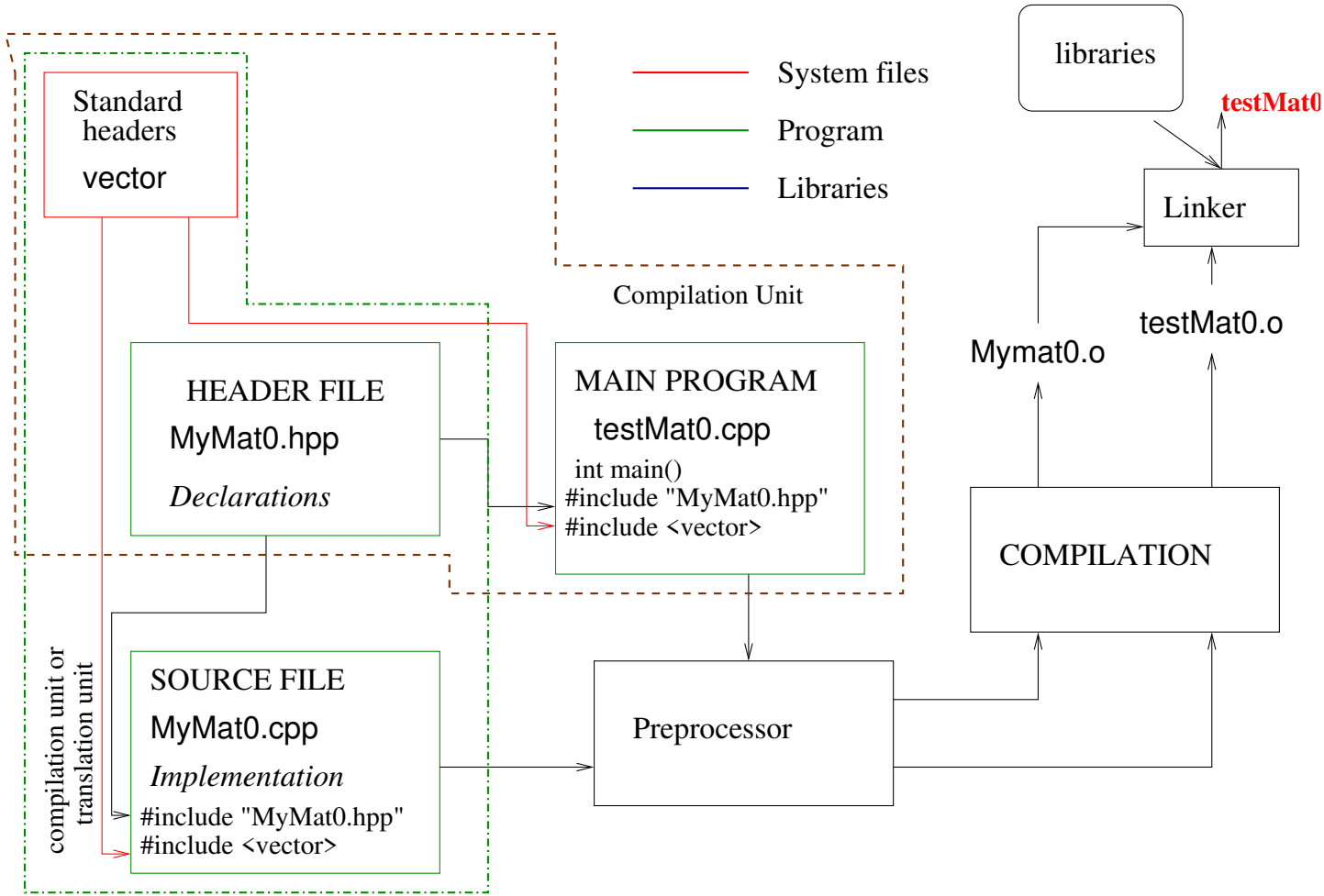
Software organization

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)

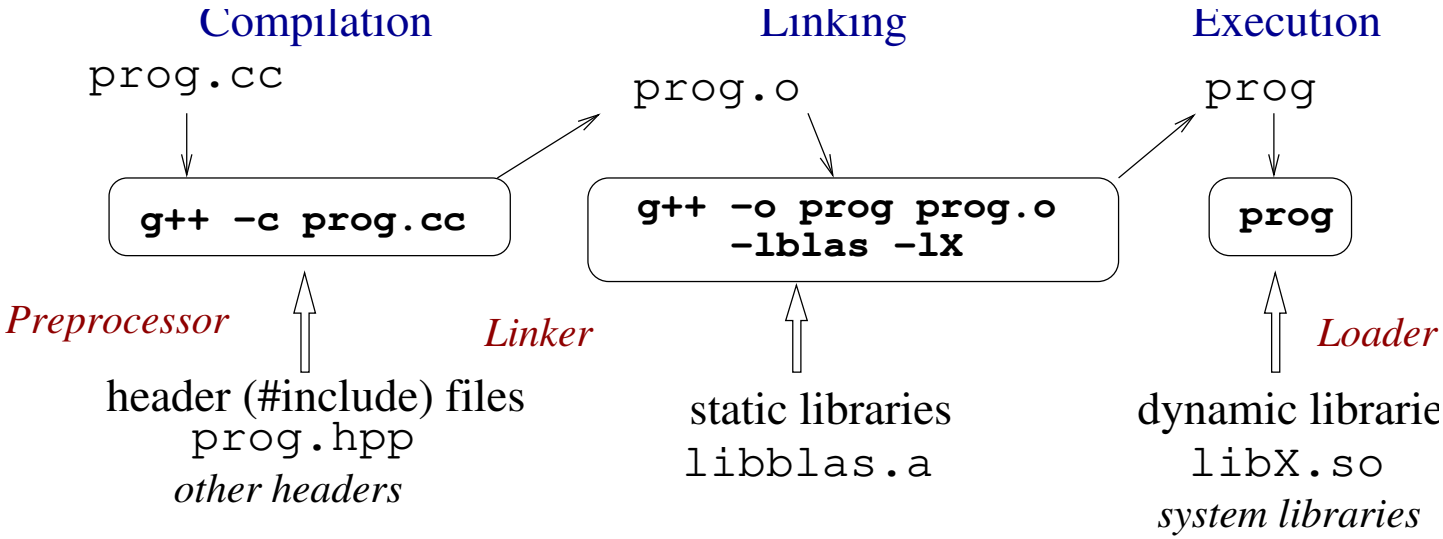
Compiling an executable is in fact a multistage process formed by different components. The main ones are

- 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-



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_MYMATO__HH
#define HH_MYMATO__HH
... Here the actual content
#endif
```

The variable after the `ifndef` (`HH_MYMATO__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).

Advanced Programming for Scientific Computing (PACS)
Lecture title: Some elements of C++

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

A note on the C++ language

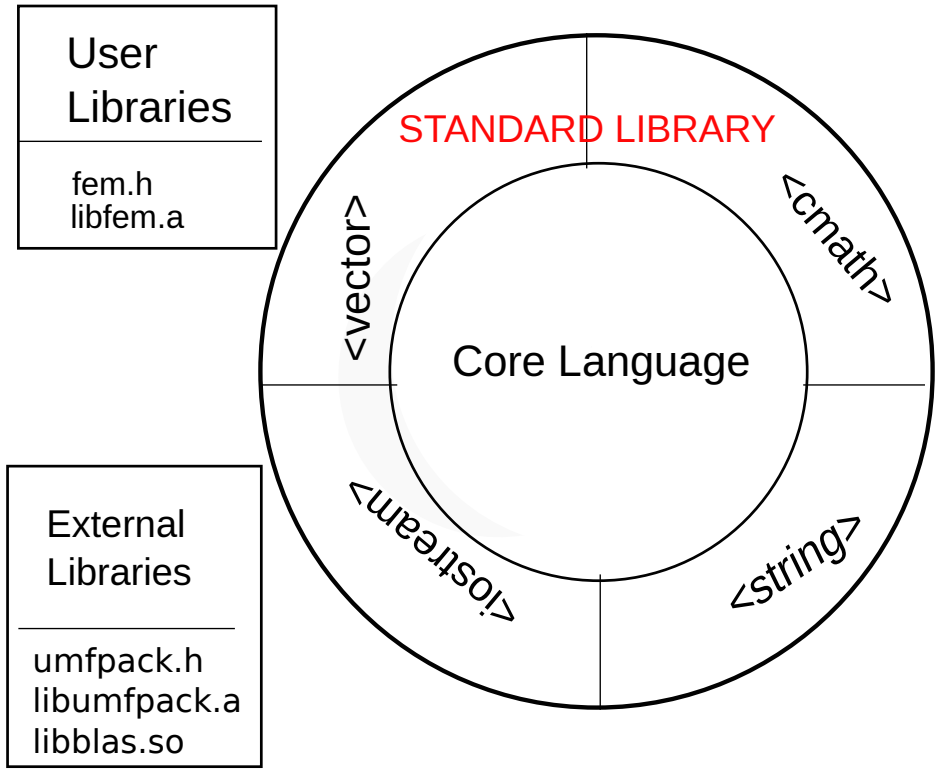
C++ has evolved recently. Big changes have been made with the `C++11` standard, further less disruptive additions in the C++14 and C++17. Important additions have been made in C++20. The oldest standard is indicated by `C++98`.

All major compilers now implement the C++17 standard fully, and good part of the C++20 standard. In this course I normally refer to C++17, but I will use also a few C++20 features.

An important feature of C++ is backward compatibility. A code written using C++11 standard almost surely compiles if you use a C++17 compliant compiler. You may indicate the standard you want with the option `-std=` of the compiler.

In particular, for the examples I use `-std=c++20`.

The structure of the C++ language



C++ is a highly modular language. The core language is very slim, being composed by the fundamental commands like `if`, `while`,.... The availability of other functionalities is provided by the *Standard Library* and require to `include` the appropriate `header` files.

For instance, if we want to perform i/o we need to include `iostream` using **#include** <iostream>.

The `main()` Program

The main program defines the only entry point of an executable program.
Therefore, in the source files defining your code there must be one and only one `main`. In C++ the main program may be defined in two ways

```
int main (){ // Code
}
```

and

```
int main (int argc, char* argv[]){ // Code
}
```

The variables `argc` and `argv` allow to communicate to `main()` parameters passed at the moment of launching.

Their parsing is however a little cumbersome. The use of GetPot utility makes parsing easier.

What does `main()` return?

In C++ the main program returns an integer. This integer may be set using the return statement. If the return statement is missing, by default the program returns 0 if terminates correctly.

The integer returned by `main()` is called *return status* and may be interrogated by the operative system.

Therefore, you may decide to take a particular action depending on the return status. Remember that by convention status 0 means "executed correctly". If the main does not have a `return n` statement, and you do not call `std::exit(n)`, the returned value is 0.

Just to let you know: if you terminate a program with a call to `std::abort()` the program aborts and return status is undefined, but the system signal `SIGABRT` is sent instead. `abort()` is a brutal way of terminating a program. Don't do it.

Two simple examples to start with

A simple C++ program. In `Examples/src/SimpleProgram/main_ch1.cpp` A program that computes $\sum_{i=n}^{m} i$, reading m and n from file.

HeatExchange. A simple 1D finite element program, where we also give an example of the use of GetPot, the json file reader, and of gnuplot-iostream, to create the plot of the solution from within the code. (maybe it's not working if you are using a virtual machine).

Some nomenclature

- Identifier An identifier is an *alphanumeric string* that identifies a variable or a function uniquely. The identifier of a variable is its *fully qualified name*, for a function is its signature, which includes the type of the function parameters and, for function members, the possible `const` qualifier.
- Symbol The translation of an identifier in the compiled code.
- Name An alphanumeric string associated to variables or sets of overloaded functions. Names in C++ cannot begin with a numeric character and are case sensitive. A qualified name contains the name of its enclosing class or namespace, using the scope resolution operator ::. A name cannot be equal to a `keyword` of the language (e.g. I cannot have a variable called **while**).

Declaration and definition

- A (pure) declaration is a statement that informs the compiler about the existence and type of a variable or object, but does not allocate memory for it. A declaration can be done at any place in the code, as long as it's done before the variable or object is used.
- A definition, on the other hand, is a declaration that also allocates memory for the variable or object. A variable or object can only have one definition (unless declared **inline**), but it can have multiple identical declarations.

In C++, variables and objects can be declared multiple times in the same or different scopes, but only defined once (ODR rule) in the entire program (unless declared **inline**).

Nomenclature: Scope

Every name in a C++ program is accessible only in some part of the source code called it's scope. A declaration introduces a name in a scope. We can distinguish

- Local scope A name declared in a function is local. It extends from its point of declaration to the end of the block in which its declaration occurs.
- Namespace scope It is a scope with a name (even if we may have anonymous namespaces!). It extends from the point of declaration to the end of the namespace.
- Class scope. A name is called *member name* when defined in a class. Its scope extends from the opening { of its enclosing declaration to the matching }.
- Block scope A set of statements included between { and }.

Facts about scope

- Scopes can be nested and an internal scope "inherits" the names of the external scope.
- A name declared in a scope *hides* the same name declared in the enclosing scope. If the enclosing scope is a namespace scope, name is still accessible using the scope resolution operator `::`.
- The most external *scope* is called global.
- The scope resolution operator `::` allows to access variables in the global scope.
- `for` and `while` define a scope that includes the portion with the test.
- A variable is destroyed when the program execution exits the scope where it has been defined.

In Scope/main_scope.cpp a small example about scope rules.

Nomenclature

- Object. With the term *object* we refer to a memory location storing something useful. The process of creating an object is called *construction*, while the process by which the object is erased from memory is called *destruction*.
- Variable A named object in a scope. It represents a specific item of data that we wish to keep track of in a program. All variables have a *type*. A variable in a class scope is a *member* of the class and accessible by all its methods (member functions). A variable is *constant* if its content cannot be changed. A constant variable is created using the `const` qualifier.
- Hiding Mechanism by which a name in an enclosed scope hides a name declared in the enclosing scope.

Nomenclature

- Redefinition. A consequence of hiding: a function/variable in a derived class (or in a nested scope) replaces a that in the base class.

```
struct B{
double fun(double);
};
struct D: public B{
double fun(double);
};
...
D d;
B b;
b.fun();// calls B::fun()
d.fun();// calls D::fun()
```

But I can do d.B::fun(); and call the fun of B!.

Nomenclature

- Overloading. Process by which functions with the same name, but different identifiers can be present in a program

```
double foo(double const &);// this foo takes a double
double foo(int); // this foo() takes a int
...
auto x = foo(5);// calls foo(int)
auto y = foo(5.0);// calls foo(const double &)

class C
{
public:
double & getValue();
double getValue() const;
}
...
C c;
const C cc;
auto& x = c.getValue();// non-const version
auto y  = cc.getValue();// const version
```

Nomenclature

- Polymorphic overriding. Process by which a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

```
class B
{
public:
virtual double fun(double);
};
class D: public B
{
public:
double fun(double) override;
}
...
B b;
D d;
B* bp = new D;
B & br = d;
bp->fun(4.);// calls D::fun(double)!
br.fun(5.0);// calls D::fun(double)!
```

Nomenclature

- POD Plain Old Data: int, double, bool,... all basic types common with the C language. More precisely, compiling a POD in C++ produces the same memory layout as in C.
- Variable Qualifiers keyword that change the behaviour of a variable: **const**, **volatile**. A variable is of a cv-qualified type if it is either **const** or **volatile**.
- Adornments References are sometimes indicated as "adornments' of a type since they do not define a new type, but provide a different name to an existing object.
- Integral type. Types that can behave like an **int**: **int**, **unsigned int**, **long int**,**short int**, **bool**, **char**, **byte**, enumerators, and all pointers. They are the only values that can be set as template parameter.

Nomenclature

- Operators An operator is a particular function that combines one or two arguments (in one case three arguments) and returns a value. E.g. + can be a unary operator, as in c=+5, or a binary operator (addition), as in c=a+b. C++ allow overloading of most operators. A full list of C++ operators is found on Wikipedia!.
- Expressions Anything that produces a value, i.e. 5+a where a is a int, is an *integer expression*. The type of an expression is the type of the value provided by the expression. An expression may be composed by subexpressions, like in (a+b)(d+e).
- Statement is a line of code that performs a specific action, usually terminated by a semicolumn (;). A program is a sequence of statements.

Nomenclature

- Constant expression. A constant expression is an expression that can be computed *at compile time*. Constant expressions may be defined with the `constexpr` specifier:

  ```
  constexpr double pi=3.1415; //pi is a constant expression
  ```

- Literal. Literals are another case of constant expressions, they are written "literally". They are used to express given values within a program: in **double** a=2.3;, the expression `2.3` is a literal (of type **double**). In std::**string** a{"Hello World"}; the expression `Hello World` is a literal (of type **char***).

Literal types

The type of a numeric literal may be specified by a suffix. By default a numeric literal containing a dot (.) is a `double`, otherwise it is an `int`. Other literal suffixes:

```
3 // int
3u // unsigned int
-123456789l // long int
12345678912345ul // unsigned long int
3.4f; // float
3.4 // double
3.4L // long double
```

Important Note: 3/4 is equal to 0: you are making an integer division! And if you do **auto** a =10;, variable a is an **int**.

Complex literals

If you include <complex> and set

```
using namespace std::complex_literals;
```

you have the complex literals

`i`, `if` and `il`

for the imaginary unit, expressed as double, float and long double respectively:

```
auto c = 4.0 + 3.0i // c is a complex<double>
auto d = 5.0L + 4.5il // d is a complex<long double>
```

String literals

String literal should deserve a lenghty discussion since strings are special in several aspects. One of which is that they are interpreted. For instance, the sequence "\\n" when printed means "line feed". If you want to not interpret special characters like \n in a string, you must use special raw string literals.

```
std::string S=R"foo(Hello\n World\n)foo";
std::cout<<S;
```

will be printed verbatim as `Hello\n World\n`.

In C++ you have std::**string** (very nice!), but also the old fashioned C-style null-terminated string (basically **char***).

C-strings and C++ std::string

A C-style null-terminated string is implicitely convertible to a std::**string**, that's why

```
std::string s="this is a string";
```

is perfectly fine, even if "this is a string" is in fact a **char** *!.
If you want an alphanumeric literal to by interpreted as a std::**string** you may use the "s" suffix:

```
using namespace std::string_literals; // you need this!
auto s="this is a string"; // s is a char*
auto S="this is a string"s; // S is a std::string
```

We will not enter into too many details. You may have a look to some string literals in the folder StringLiterals.

Fixed width integer types

With the proliferation of different architecture sometimes it may be necessary to write safe code to specify exactly what we mean with **int**: is it a 16 bit or a 32 bit integer? The header <cstdint> defines integer types guaranteed to have a specific width.

```
#include <cstdint>
int_8_t a; // 8 bits integer
int_32_t b; // 32 bit integer
uint_64_t c;// 64 bits unsigned integers..
```

And there are many others: take a look here.

If lives, or lots of money depend on your code... it is better to know which type of integer you are dealing with! Remember that integer overflow is difficult to detect.
Look at the small code in IntegerOverflow.

Instance and initialization

- Instance. The moment when an object is created (the meaning is different for templates).
- Initialization. To provide the initial value to an object during its construction.
- Assignment. To provide a value to an already constructed object.

Initialization and assignment

```
vector<double> v={2,3,4};// v is initialized to a vector of 3 elem
double a{3.5};// a is initialized
double b=0.5;// b is initialized
b=a; // the value of a is assigned to b
vector<double>z{v};// z is initialised with the value of v
double k; // k is not initialized
```

because b was already created

is always better to initialize variables

Beware that **double** a=b is an initialization: a is created by copying the value contained in b (copy-construction). While b=a is an assignment: the value of the existing variable a is changed by copying the value in b (copy-assignment).

Initializations of the form **double** a{3.5}; or (old style) **double** a(3.5); are called direct initializations. They are almost equivalent to **double** a=3.5; (copy initialization), the difference concerns conversion rules (we will discuss this issue later).

In-class definitions and initialization

Methods can be defined "in-class". Non static data members may be initialised in-class and also static data by declaring them **inline**;

```
class MyClass
{
    ...
    public:
    // in-class definition
   double get_x() const {return x;}
   double a{5.0}; // in class initialization
   inline static float c=6.0f;// in class initialization
};
```

as in general static variables could not be initialized in class

Definitions, when not inline, templates or in-class go in source files.

An important note

Don't assume that a variable is initialized automatically. Initializing variables explicitly is safer!.

Always initialize pointers to a value. If the object is not available at the moment, initialize pointers to the null pointer. Never have dangling pointers in a program.

```
double * p=0; // OK. A null pointer. But prefere nullptr
double * x=nullptr;// a null pointer (much better!)
int * ip=new int[20]; //Ok pointer to a C-array of int
double * pp;// NO!NO!NO!
```

A reference must always be initialized (bound ia a more precise term) to an existing object!

Brace ({}) initialization

We will give more details in the uniform (brace) initialization in a later lecture. Here I just recall that in general we can initialize variable with braces or parenthesis

```
double x1(20.); //Ok x value is 20.
double x2{20.}; //Ok x value is 20.
float x3(std::sqrt(10.)); // ok double converted
float x4{std::sqrt(10.)};// ERROR: narrowing not allowed
double x5{};// Ok initialized by 0
double x6();// NO! it's a function decl.
double x6={};// Ok, initialized by zero
double x7=();// NO! It does not make sense
```

loss of precision (like here double to float)

In modern c++ brace initialization is preferred.

Note: **double** a{}; initializes a by zero, while **double** a; leaves it uninitialized.

Formatted i/o

C++ provides a sophisticated mechanism for i/o through the use of streams. We will see more details later on. For the time being, we recall that streams may be accessed by the `iostream` header file.

```
#include <iostream>
..
 std::cout <<''Give me two numbers'' << std::endl;
 std::cin >> n >> m;
```

The standard library provides 4 specialized streams for i/o to/from the terminal.

std::cin	Standard Input (buffered)
std::cout	Standard Output (buffered)
std::cerr	Standard Error (unbuffered)
std::clog	Standard Logging (unbuffered)

for optimization may not be immediately executed

cerr and clog

`cerr` and `clog` are by default addressed to the standard output. But, you can redirect them either at operative system level (with the exception of `clog`),

```
./myprog 2>err.txt #redirecting stderr to a file
./myprog &>allout.txt #redirect both stderr and stdout
./myprog 1>out.txt 2>err.txt #to different files
./myprog &> output.txt #both stderr and stdout to a file
```

or internally in the program (we will see how to do in a lecture dedicated to input output).

Unbuffered means that the stream is immediately sent to the output, with no internal buffer. The internal buffer is used to make i/o more efficient, but it may be deleterious for error messaging, since if the program fails the message may not be printed out.

Implicit and explicit conversion

```
int a=5; float b=3.14;double c,z;
c=a+b (implicit conversion)
c=double(a)+double(b) (conv. by construction)
z=static_cast<double>(a) (conv. by casting)
```

also with double{a} or here we are calling the ctor for double

C++ has a set of (reasonable) rules for the implicit conversion of POD. The conversion may also be indicated explicitly, as in the previous example.

Note: It is safer to use explicit conversion, but implicit conversions are very handy! Yet, if you want to make your intentions clear use explicit conversions.

C-style cast, e.g. z= (**double**) a;, is allowed but discouraged in C++. Don't use it. **static_cast** is safer.

love at compile time ⇒ safer

Casts

Casting is an expression used when a value of a certain type is explicitly "converted" to a value of a different type. In C++ we have three types of cast operator (well, 4 if we count also C-style cast)

1. **static_cast**<T>(a) is a safe cast: it converts a to a value of type T only if the conversion is possible (in the lecture on classes we will understand better what it means). For instance **static_cast**<**double**>(5) is possible (but unnecessary since conversion is here implicit) but **static_cast**<**double** *>(d), when d is a double, gives an error because there is no way to convert a double value to a pointer to double value.
2. **const_cast**<T>(a) removes constness. It is a safe cast as well.
3. **reinterpret_cast**<T>(a). This is an unsafe cast, to be used only when necessary and with extreme care. It takes the bit-wise representation of a and interprets it as if it were of type T. It is up to you ensuring it makes sense: no checks are made. There are limitations on its use, but not much relevant.

the using keyword and template alias

```
using Real=double; // equivalent to typedef double Real
// func is a pointer to function double->double
 using func = double (*) (double);
// a function taking a function as argument
double integrate(funct f, double a, double b);
//usage
Real a; // Defining a double
// integrate the sin();
auto result=integrate(std::sin,0.,10.);
```

Prefer **using** to the old style **typedef**.

For functions, we will see that we have a much better option than pointers to function!

A suggestion

Use **using** to define aliases to types that are either complicated, or that you may change in the future, or just to give a more significant name, easier to recall

```
// maybe in the future I may want to use float instead
using Real=double;
// This is my choice for Vectors
using Vector=std::vector<Real>;
// To simplify life
using MapIter=std::map<std::string,std::string>::iterator;
...
```

Types

We recall that C++ is a strongly typed language. Any variable and any expression has a type and the type must be known at compile time.
You effectively define a new type every time you declare a class, struct or and enum and every time you instantiate a class template.

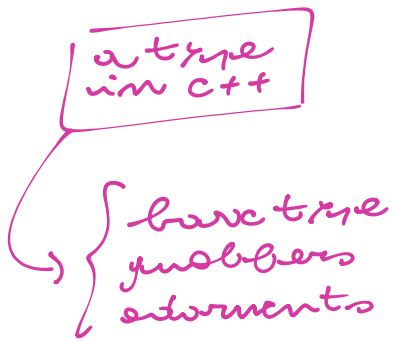
```
class ParametricModel{
  ...};// introduces the type ParametricModel

std::vector<double> a; // a is of type vector<double>,
// a particular instance of a vector<T>
```

However, we can let the compiler *automatically deduce* the type. This possibility is provided by **auto**, decltype() and decltype(**auto**) specifiers.

What is a type?

Before dwelling into **auto** and decltype(**auto**) it is important to understand that a type in C++ is in fact formed by different components:

- The "basic type", like **int** or std::vector<**double**>, which gives information on how the object should be interpreted (and the size it uses up in stack memory). This is indeed the actual type of an object;
- The possible qualifiers: **const** or **volatile** that define the type of access: **const double** a indicates that a is "read only";
- "Adornments" which indicate that the variable is an "alias" to an existing object. They are the l-value and r-value references: **double** & b=a, **const double**&& c=a, here b and c are alternative secondary names for a. (if you don't know what a r-value reference is, don't worry, we will see them later on).



In **const double** & z=a I am defining a reference to a constant double. The "basic" type is **double**. The type is qualified as **const**, so I cannot change a through b.

the auto keyword

In the case where the compiler may determine automatically the type of a variable (typically the return value of a function or a method of a class) you may avoid to indicate the type and use **auto** instead.

```
vector<double>
  solve(Matrix const &,vector<double> const &b);
vector<int> a;
...
auto solution=solve(A,b);
// solution is a vector<double>
auto & b=a[0];
// b is a reference to the first element of a
```

auto returns the base type, omitting qualifiers and adornments, i.e. you must add & or/and const yourself if you need them.

Some warnings

auto is a very nice feature. However it may sometimes make the program less understandable, so declare the type explicitly when you think it helps readability.

Beware, moreover, that not always things are what they seem....

```
std::vector<bool> a; // a vector of boolean
...
auto p = a[3]; // p IS NOT A bool!
```

A vector<**bool**> is treated specially: the boolean values are packed inside the vector (one bit per boolean). The returned value of [] operator is a special proxy to a `bool`, convertible `bool`. But in fact you would like to have a full-fledged bool for `p`, so you should avoid auto in this case:

```
bool p = a[3]; // Avoid auto in this case
```

Other uses of auto

We will see other uses of auto in the lecture about functions and lambda expressions.

Extracting the type of an expression

You are probably aware of the command **sizeof**(), which returns the size of a type (in bytes). For instance **sizeof**(**double**) returns 8 (in most systems). It can be applied also to expressions: **sizeof** a+b; returns the size of the type returns by the expression. We can interrogate also the type of an expression using decltype()

```
const int& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1; // const int& X1
decltype(i) x2; // int x2
decltype(a->x) x3; // double x3
```

it returns the exact type (bare type, qualifiers and ampersands)

This new feature can be useful in generic programming.

decltype(**auto**)

decltype(**auto**) is a different form of **auto** that has different type deduction rule.
We will see its usage in the lecture on functions.

Pointers

Pointers are variables that store the address of an object, and enable us to get the object they point to by dereferencing the pointer (with the exception of pointer to **void** whose discussion is postponed). This code

```
double x =5;
double * px= &x; // get the address of x
std::cout<< *px; // dereferencing px
```

prints 5.

A special pointer called null pointer, and indicated by nullptr (or simply 0), indicates that the pointer is not containing any valid address (it points to nothing). A null pointer contextually converts to **false**, while a non-null pointer converts to **true**. So the statement **if** (px !=nullptr) is often written just as simply **if**(px).

Pointers, smart and not

In modern C++ we can use pointers for different purposes: to handle resources dynamically or to implement polymorphism, for instance.

For handling resources dynamically, in modern C++ it is much better (almost mandatory!) to use smart pointers.

We have a specific lecture on smart pointers.

Constant variables

The type qualifier **const** indicates that a variable cannot be changed, thus a constant variable must be initialized with a value (with an exception that we will see later).
Since C++11 we have the keyword **constexpr** to indicate constant expressions whose value is known at compile time.

```
double const pi=3.14159265358979;
double const Pi=std::atan(1.0)*4.0;
const unsigned ndim=3u;
double constexpr Pi=3.14159265358979;
float constexpr E(2.7182818f);
double constexpr PI=std::atan(1.0)*4.0;// NOT POSSIBLE
constexpr unsigned Ndim=3u;
```

It is better to understand the difference.

Setting away **const**

So the main difference of const wrt constexpr is that actually const declared variables could be changed. While constexpr variables surely not.

Let's recall that in fact you may still change the value of a constant variable using the special cast operator called **const_cast**.

However, if you are obliged to use **const_cast** it means that your code IS POORLY DESIGNED!.

const_cast is only for emergency situations, typically if you have to interface with poorly written or C code, see the next slide.

A case for const_cast<T>

Real world in imperfect. We may have the necessity of stripping the **const** attribute, for instance to be able to call legacy functions where the author forgot to use **const** to indicate arguments that are not changed.
In this case we may use **const_cast**<T>().

```
// this function does not change a and b but by mistake
// they are  taken as double & and not const double &
double minmod(double & a, double & b);
...
// A function that uses minmod
double fluxlimit(double const & ul, double const& ur){
...
minmod(const_cast<double>(ul),const_cast<double>(ur));
```

const vs. constexpr

While a const variable is indeed a variable a **constexpr** is not really a variable.
The compiler is free to not allocated memory for it and use its value directly at compile time:

```
constexpr  double a=5.0;
double const b=a*a; // The compiler may compute 25
```

Constant expressions are immutable. There is no way to change their value.
Integral constexpr may be used as tempalte parameter values.

```
constexpr int N=10;
std::array<double,N> arr;// I can use it as template argument
```

is exactly like

```
std::array<double,10> arr;
```

const vs constexpr, concluding remarks

In conclusion,

- Use **const** to indicate that a variable is not meant to change. You have more efficient and safer code!
- Always declare **const** methods that do not change the state (i.e any variable member) of a class;
- Use **constexpr** to indicate constants (like π or e) that are immutable, or integers that you may use as template argument. The compiler may use them directly at compilation time, producing a more efficient code.

Note: C++20 provides the header <numbers> with a lot of useful numeric constants!.

const and constexpr are your friend!

Use them!. If function parameters are references to something that is not changed inside the body of the function they must be declared const, it is not an option!.

```
void LU(Matrix const & A, Matrix & l, Matrix & u);
```

Here `A` is not changed by the function, so we must declare it `const`.

Remember that a non-const references doesn't bind to a temporary object:

```
LU(CreateBigMatrix(),L,U);
```

won't work if A had been declared a non-const reference! While a const reference can bind to a temporary.

A Note: Here, when use the term reference alone I mean l-value references (we will discuss the && stuff later on.)

constance rules

`const` (and `constexpr`) are associated to the item on their left, unless they are the first keyword in the type definition, in which case they apply to the item on the right. The statement

```
double const * const p;
```

means *p is a constant pointer to a constant double*. Neither the value of the pointer nor that of the pointed object can be changed!. While,

```
double const * p;
```

is a pointer to a constant double. You can change the pointer, but not the value!

Another Note: A reference is always const (reference cannot be reassigned). So **double** & **const** a is simply wrong! However, often the term *const reference* is used to indicate a reference to a constant object (and I normally use this abuse of language).

Enumerators

I assume that you already now about enumerators:

```
enum bctype {Dirichlet,Neumann,Robin};\\definition
...
bctype bc;
..
switch(bc){
 case Dirichlet:
...
 case Neumann:
...
 Default:
...
}
```

=0 =1 =2

we could also write that, like a dictionary

They are just "integers with a name", implicitly convertible to integers.

Scoped enumerators

The implicit conversion to integers may be unsafe. In C++ we also have scoped enumerators that behave as a user defined type and are not implicitly convertible to int (you need esplicit casting).

```
enum class Color {RED, GREEN, BLUE};
...
```

we add class to remove the implicit conversion to an integer

```
Color color = Color::GREEN;
...
if (color==Color::RED){
    // the color is red
}
```

Now the test **if** (color==0) would fail, but you can still do explicit conversions, if needed, **int** ic=**static_cast**<**int**>(color);

RVO, return value optimization, or copy elision

Let's finish this long lecture illustrating an important (and old) optimization that saves memory and time. It is called copy elision or return value optimization. Let's have a function that creates a matrix, returning it by value (as it should be!).

```
Matrix createBigMatrix();
```

And a class SVD that stores a Matrix to do some operations, for instance its SVD decomposition.

```
class SVD{
  Matrix My_A;
  public:
  SVD(const Matrix & A):My_A{A}{};
}
```

Copy elision

Now, have a look at this code,

```
SVD svd; // performs svd
Matrix A=createBigMatrix();// initialize a by copy
SVD svd(A);// pass A to SVD constructor
```

Now I have the matrix in A and the one stored in svd. But maybe I do not need matrix A anymore and I am wasting memory, as well as time since I am copying a big object.

The correct solution is:

```
SVD svd{createBigMatrix()};
// or
SVD s = createBigMatrix();
```

We directly pass the output of the fct to create svd or s => direct construction

This activates copy elision: the matrix returned by the function is directly used to construct the matrix stored in svd. No memory waste! No useless copies.

Move semantic

We will have a lecture on move semantic, where I will describe other ways of saving memory, an important issue if you are dealing with “big data”.

Advanced Programming for Scientific Computing (PACS)
Lecture title: A brief introduction to floating point numbers

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Floating point number system

A normalized floating point number system F is formed by *zero* and real numbers of the form

$$y = \pm m \times \beta^{e-t}$$

where

β	basis (typically 2)	t	precision
m	mantissa, $\beta^{t-1} \leq m < \beta^t$	e	exponent, $e_{min} \leq e \leq e_{max}$

Or,

$$y = 0.d_1 d_2 ... d_t \times \beta^e \quad d_1 \neq 0, \quad 0 \leq d_i < \beta$$

we use all the space for the digits, we don't use one more precision

All floating point numbers representable in a computer belong to $F^* = F \cup F_s$ where F_s is the set of *subnormal numbers* of the form

$$y = \pm m \times \beta^{e_{min}-t}, \quad 0 < m < \beta^{t-1}$$

The values of t, e_{min} and e_{max} characterize the type of floating point number and is defined by the IEEE 754 standard.

Round off

A real number $x \in \text{range}(F^*)$ is approximated in a computer by $\hat{x} = fl(x) \in F^*$ and $e_r = (x - \hat{x})/x$ is the (relative) rounding error.

If $x \in \text{range}(F^*)$ then $\hat{x} = fl(x) \in F^*$ and $e_r = 0$. In general,

$$|e_r| \leq u = \frac{1}{2}\beta^{1-t}$$

where u is the roundoff unit. The machine epsilon ϵ_M is the smallest positive floating point number by which $fl(1 + \epsilon_M) \neq 1$. We have:

$$u = \frac{1}{2}\epsilon_M$$

IEEE arithmetic

The IEEE 754 has been defined in 1985 (and amended various times since then) and defines the floating point arithmetic system normally implemented in modern processors. There are two main types of floating point numbers

Type	Size	t	e	u	Range
float	32	$23+1$	8	2^{-24}	$10^{\pm 38}$
double	64	$52+1$	11	2^{-53}	$10^{\pm 308}$

The value in the table indicate the number of bits used. Moreover, the implementation of IEEE arithmetic system should satisfy the standard model: if $x, y \in F$ then

$$fl(x \,\mathrm{op}\, y) = (x \,\mathrm{op}\, y)(1+\delta) \quad |\delta| \leq u, \quad \mathrm{op} = + \; - \; \times \; /$$

Special numbers

The IEEE standard prescribes that

$$fl(x) = 0 \quad \text{if } |x| < F_{min} \quad \text{(UNDERFLOW)}$$

$F_{min} \in F^*$ being the smallest positive floating point number.

$$fl(x) = \text{sign}(x)Inf \quad \text{if } |x| > F_{max} \quad \text{(OVERFLOW)}$$

$F_{max} \in F$ being the maximum floating point number.

Moreover, $x/0 = \pm Inf$ if $x \neq 0$, $Inf + Inf = Inf$ and $x/Inf = 0$ if $x \neq 0$.

Finally, the special number NaN (Not-a-Number) indicates the result of an invalid operation ($0/0$, $\log(-1)$ etc) and we have $x \,\text{op}\, NaN = NaN$, $Inf - Inf = NaN$ e $Inf/Inf = NaN$.

Floating point exceptions

The standard prescribes that in normal situations an invalid operation or an over/underflow do not stop computations, but produces the special numbers illustrated before.

However, the processor may record if an invalid floating point operation (normally called floating point exception) has occurred, so that the user may trap it.

We will discuss this issue into more detail later in the course, showing how you can capture floating point exceptions.

Forward and backward error

Let $f : \mathbb{R} \to \mathbb{R}$ and $\hat{f} : F \to F$ the corresponding expression on floating point numbers. Let $y = f(x)$ e $\hat{y} = \hat{f}(\hat{x})$. The analysis of the forward error aims to find a δ such that

$$e_f = \frac{|y - \hat{y}|}{|y|} \leq \delta$$

The relative *backward error* Δ is defined by $\hat{y} = f(x(1 + \Delta))$. If $f \in C^2$ we have that

$$\frac{|y - \hat{y}|}{|y|} = c(x)|\Delta| + O(\Delta^2)$$

with $c(x) = |xf'(x)/f(x)|$ called the relative condition number of f. In general, one looks for an estimate such that

$$e_f \leq C(x)|\Delta|$$

A simple example: cancellation

Let us consider $y = f(a, b) = a - b$ e $\hat{y} = f(a(1 + \Delta), b(1 + \Delta))$. It is easy to find out that

$$|\frac{y - \hat{y}}{y}| = \frac{|a\Delta - b\Delta|}{|a - b|} \leq \frac{|a| + |b|}{|a - b|}|\Delta| \Rightarrow C = \frac{|a| + |b|}{|a - b|}$$

We have that $C \to \infty$ when $|a - b| \to 0$: the subtraction of almost equal floating point values causes a big round-off error. We should avoid it whenever possible!

A thorough analysis of floating point errors for common mathematical operations is found in the book by N.J. Higham Accuracy and stability of numerical algorithms, SIAM, ISBN: 978-0-89871-521-7, 2002.

An example: numerical differentiation

If $f(x) : \Omega \to \mathbb{R}$ is sufficiently regular we have

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2), \quad \forall x \in B_h(x).$$

Therefore the centered formula $\delta_f(x) = (f(x+h) - f(x-h))/2h$ is a second'order approximation of the derivative. However, the quantity actually computed is

$$\widehat{\delta}_f(x) = \frac{f[(x+h)(1+\Delta_1)] - f[(x-h)(1+\Delta_2)]}{2h},$$

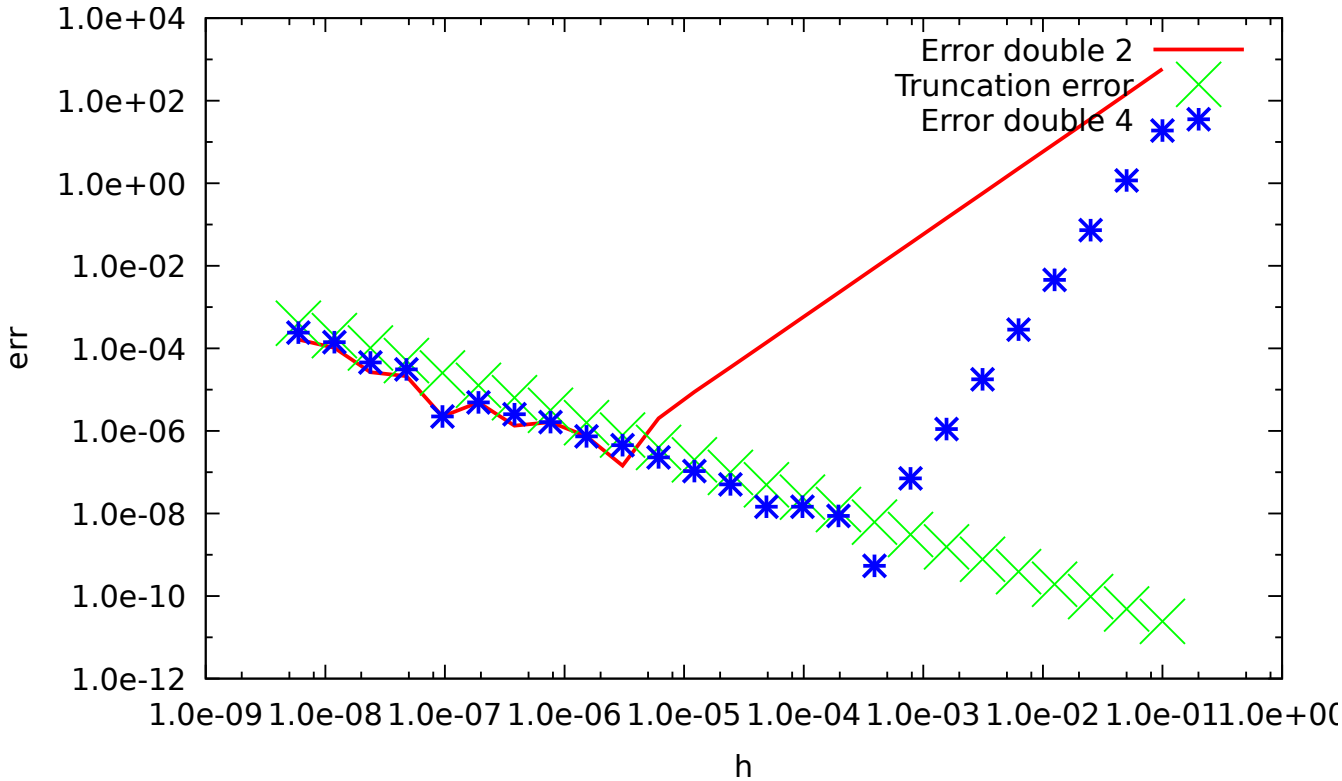
with $|\Delta_1| \leq u$ and $|\Delta_2| \leq u$. If $f'(x) \neq 0$ we may obtain the estimate

$$|\widehat{\delta}_f(x) - \delta_f(x)| \leq \frac{C}{2h}$$

with $C \leq |x||f'(x)|u$.

In the example Examples/src/FloatingPoint/FinDiff we also implement the fourth order formula $(\frac{1}{12}f(x-2h) - \frac{2}{3}f(x-h) - \frac{1}{12}f(x+2h) + \frac{2}{3}f(x+h)/h$ and we compute the numerical derivative of $100e^x$ at the point $x = 3$ with the two formulas and plot the error $|\widehat{\delta}_f(x) - f'(x)|$ for different values of h, together with the estimated forward truncation error. We repeat the example using single, double and extended precision

The result for double precision



You may note that for $h \lesssim 10^{-6}$ the truncation error dominates the second order formula (Error double 2), while it already spoils the result for $h \lesssim 10^{-3}$ if we use the fourth order approximation!

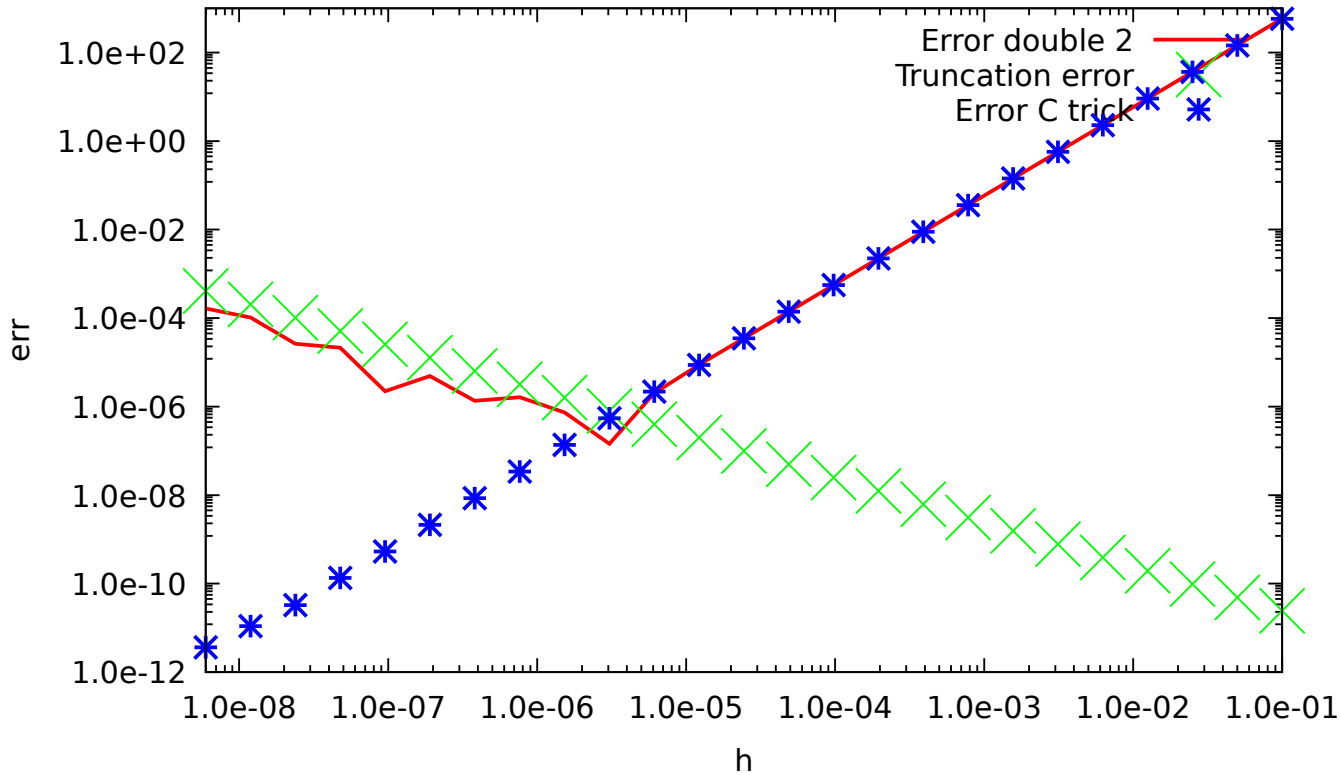
Differentiation with(out) a difference

Sometimes one may use non-standard techniques if a great precision is needed. One may for instance note that if f is extended to a complex function and if the extension is analytic we may write (we need to apply Cauchy-Riemann equation and note that $f(x)$ is real)

$$f(x + ih) = f(x) + ihf'(x) + O(h^2).$$

So $\mathrm{Im}(\frac{f(x+ih)}{h})$ is a second order approximation of $f'(x)$ and its computation does not imply taking differences!.
Indeed the result obtained for small h may be very precise. In FloatingPoint/FinDiff we have implemented also this formula. We show the result.

The result for double precision



You may note that with this trick (blue stars) we can reduce considerably the size of h. For larger h, as expected, we get results similar to the 2nd order formula.

Other nasty and less nasty examples

Some examples of floating point failures are in the FloatingPoint directory of the examples, with a file containing the explanation.

Some of them they are specially hand-crafted examples to highlight some possible unwanted side-effects of floating point operations. Normally the situation is not that bad!

For instance, in FloatingPoint/QuadraticRoot, where we show how the classic formula for the zero of a quadratic $ax^2 + bx + c$ may give incorrect results when $|b| >> ac$, because of cancellation errors.

Finally, in FloatingPoint/FPFailure you find an example that shows some "floating point failures".

Numeric limits

C++ allows you to interrogate the characteristics of the numeric types as implemented in your machine.
Not just floating points, but also integral types. In particular, we may look for the machine epsilon or the maximal representable number.
We need to use the Standard Library header `<limits>`:

```
#include <limits>
float eps=numeric_limits<float>::epsilon();
```

Since S++20 including the header `<numbers>` we have some mathematical constants in C++ style
See a complete example in Examples/src/Numeric_Limits

Beware of floating point comparisons!

Floating points have discrete values. So comparing them can be very critical. The statement

```
double a,b;
... // many computations involving a and b
if(a==b)....
```

is dangerous. Maybe the test is never satisfied. A stupid example

```
 double a = std::pow(3.0,1./5);
 double b = a*a*a*a*a;
 double c =3.0;
if(b==3.0) .. // IS FALSE!
```

It is better to write

```
if(std::abs(a-b)<tol)...
```

where `tol` is a well chosen tolerance (unfortunately it's not always evident what "well chosen" means.)

Advanced Programming for Scientific Computing (PACS)
Lecture title: Basic containers: vectors, arrays, tuples

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

C++ Arrays and Vectors

With arrays normally we indicate a data structure with a linear and contiguous representation of the data. In C++ we have different type of arrays

- C-style fixed-size arrays derived from C: **double** a[5], **float** c[4];
- C-style dynamic arrays, through pointers: **double** *p=**new double**[N]
- The vector container of the standard library: std::vector<**double**> c, which supports dynamic memory management.
- Fixed size arrays of the standard library. std::array<**double**,5> c

A warm suggestion: use arrays and vectors of the standard library: fewer headaches. And they can be interfaced with their C cousins.

std::vector<T>

The standard library, to which we will dedicate an entire lecture, provides a set of generic containers, i.e. collections of data of arbitrary type. The main one is probably `std::vector<T>`.
It is a class template that implements a dynamic array with contiguous memory allocation. To use it, it is necessary to include the header `<vector>`.

it is hence cache-friendly for the sequential access

Computational complexity of main operations on vector<>

Random access	O(1)
Adding/deleting element to the end	O(1)[1]
Adding/deleting arbitrary position	O(N)

we on the small scale we usually wins over other in-theory-better structure

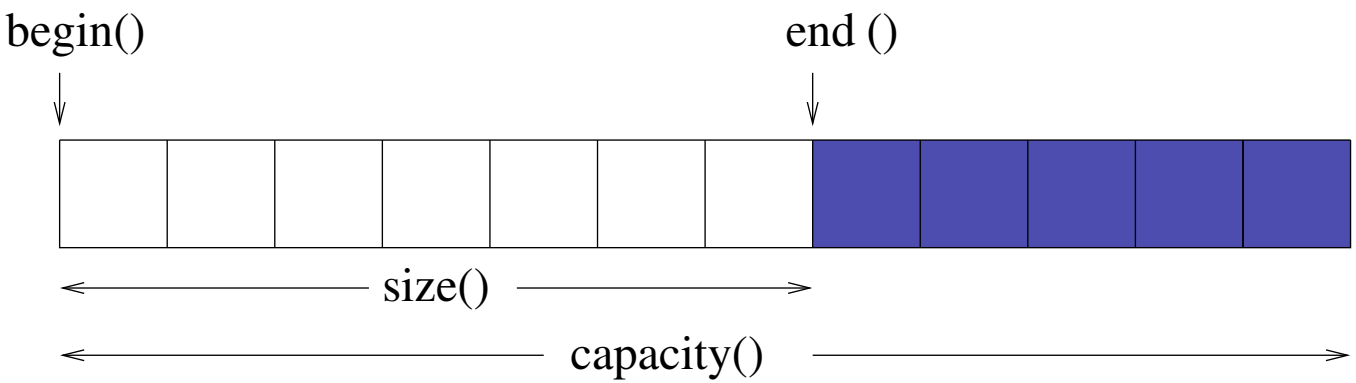
Note: we normally write vector<T> to remember that standard vectors have a compulsory template argument. But in fact the template arguments are 2!. The second is the allocator, which has a default value and is rarely changed.

[1] If the capacity is sufficient

The structure of a vector

Here you have a cartoon of the internal structure of a standard vector. For a standard array it is similar, but of course capacity is always equal to size and the size is known at compile time and unchangeable.

le "dimensioni" non variano come invece lo valore memorizzato, non esiste altro



The C++ standard guarantees that elements of vectors and arrays are contigous in memory. This ensures high efficiency.

Examples of vector<>

```
vector<float> a;//An empty vector
```

Both *size* and *capacity* is 0.

```
vector<float> a(10);//creates a vector with 10 elements
```

Here elements are created with the default constructor, in this case `float()`. `size()` is equal to 10, `capacity()` is ≥ 10. (maybe 10).

```
//vector of 10 elements initialized to 3.14
vector<float> a(10,3.14);
```

Here the elements are initialised with `3.14`. Size is 10, capacity at least 10.

```
//A vector with two elements = 10 and 3.14!!
vector<float> b{10,3.14}; // initializer list
```

Size is 2 and capacity at least 2.

Automatic deduction of template parameters

C++17 has introduced automatic deduction of template parameters. We will deal with this later in the course. I want just to mention that thanks to this new feature one can do

```
std::vector v={10,20}; // v is a vector<int> of 2 elements
std::vector a={30.,40.,-2.};// a is a vector<double> of 3 elements
std::vector c={1,2.3};//COMPILATION ERROR! Ambiguous!
```

This is indeed a nice simplification for short vectors.

The last case is ambiguous since the compiler cannot tell if you wanted a vector<**int**> or a vector<**double**>. Not being psychic, it gives up with an error.

push_back(T const & value)

The `push_back(value)` inserts a new `value` at the end (back) of the vector. Memory is handled in the following way, where *size* is the dimension of the vector before the new insertion.

1. If *size*=*capacity*
   a. allocate a larger capacity (usually twice the current one) and correct *capacity* accordingly;
   b. copy current elements in the new memory area;
   c. free the old memory area;
2. Add the new element at the end of the vector and set *size*=*size*+1;

emplace_back(T... values)

push_back: creates the element, then adds it
emplace_back: directly does the two steps

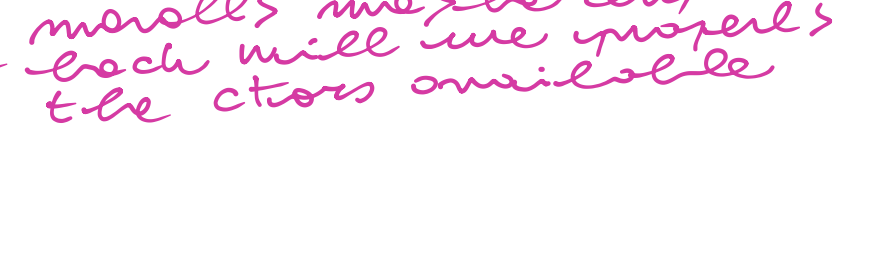
This method avoids the need of making copies when adding elements to a vector. It is sufficient to know that with emplace_back we may pass arguments to the constructor of the element directly, so the stored object is constructed in memory, with computational savings.

For the rest, it operates similarly to push_back(), so it adds a new element at the back of the vector.

Suggestion: Use emplace_back() in any case, it supersedes push_back().

Usage of emplace_back

```
class MyClass{
public:
myclass(const double, const unsigned int);// construtor
...};
...
vector<MyClass> myClassElements;
for ( std::size_t i=0;i<5;++i)
  myClassElements.emplace_back(5.0,i);
```



emplace_back(5,i) inserts a new value by calling the constructor of MyClass that takes a **double** and an **int** as arguments.

Important note: if MyClass has no default constructor (it is not default-constructible), I cannot use push_back() with no arguments and if it is not copy-constructible I can't use push_back(value) with value an object of type MyClass.

While, I can still use emplace_back(Args...) ! using the available constructors.

Addressing elements of a `vector<T>`

Elements of `vector<T>` can be addresses using the array subscript operator `[]` or the *method* `at()`. The latter throws an exception (*range_error*) if the index is out of range, i.e not in within $[0, size()[$.

```
vector<double> a;
b=a[5]; //Error (a has zero size)
c=a.at(5)//Error Program aborts
// (unless exception is caught)
```

Beware: testing vector bounds is expensive! Use it only for debugging or if necessary.

Which is the type of an index?

The index used to address a vector element is an unsigned integral type. But exactly what? An **unsigned int**? or a **unsigned long int** ?.... To avoid possible mistakes, it's better not to guess. A vector<T> knows the type used to address its element. It is stored in vector<T>::size_type. It is usually equal to std::size_t, which can be used in alternative.

```
vector<double> a;
...
for(vector<double>::size_type i=0;i<a.size();++i){
  ....
  }
```

is also architectural dependent (32, 64, ... bit)

or you may use std::size_t (simpler):

```
for(std::size_t i=0;i<a.size();++i)...
```

Alternative: use iterators or a range-based for loop (see later).

Reservation, please

```
std::size_t n=1000;
vector<float>a;
for (i=0;i<n,++i)a.push_back(i*i);

vector<float>c;
c.reserve(n);// reserves a capacity of 1000 floats
// vector is still empty, you may use push_back!
for (i=0;i<n,++i)c.push_back(i*i);

vector<float>d;
d.resize(n);\\ resizes the vector
// Now I can use [] to address the elements
// and fill them  with values
for (i=0;i<n,++i)d[i]=i*i;
```

this does not change the size

this does not change the capacity

The second and third techniques are more efficient than the first because they avoid memory allocations/deallocations.

Resizing and reserving

With resize(size_type) we change the size of the vector and the possible new elements are initialized with the default constructor. resize() may take another argument which will be used for the initialization: e.g. a.resize(100,0.3). In that case the elements are initialised with 0.3.

reserve(size_type) instead only allocates the memory area. The vector does not change size!. To add elements we need to use push_back() or emplace_back().

Note: Reserve memory whenever possible: you get a more efficient code. If the size of the vector is know and does not change, consider using std::array instread of std::vector.

Shrinking a vector

Sometimes it may be useful to shrink the capacity of a vector to its actual size.

```
vector<double>a;
...// I do something with the vector
a.clear(); // Empties vector but does not return memory!
// After a clear() size is zero but capacity is unchanged
// Now I want to shrink it
a.shrink_to_fit() // Now capacity is zero
```

we still occupies memory

To swap two vectors you may use std::swap() or the method swap():

```
a.swap(b);//swaps a and b
std::swap(a,b);// swap again
```

Iterators

Iterators offer a uniform way to access all Standard containers. Moreover, they are used heavily by standard algorithms (we will see std algorithms later). One may think iterators as special pointers. Indeed they can be dereferenced with the operator * and moved forward by one position with ++.

```
vector<double>a;
...
for (auto i=a.begin(); i!=a.end(); ++i) *i=10.56;
// all elements are now equal to 10.56
```

`begin()` and `end()` return the iterator to the first and last+1 element of the vector, respectively.

Iterators versus classical loops

We could have operated in a more classical way

```
for (std::size_t i=0; i!=a.size();++i)a[i]=10.56;
```

Using iterators may have some advantages:

- Efficiency: de-referencing an iterator may be faster than the access operation (but it is not a great deal);
- Generality: the same line of code may be used for all standard containers, while `[]` operates just on `vector<>` (and `array<>`).

We will give more information on iterators in the lecture on the standard library. We just note that for `vector<>` iterators we have also the addition and subtraction operators with an integer, with the obvious meaning.

Range based for-loops

There is in fact an easier way to access all elements of a container. We can write the for loop in the previous example as

```
for (auto & i : a) i=10.56;
```

auto i : Container generates a variable (i) that will hold the value of the elements in succession.

BEWARE: you need to use **auto** & if you want to change the entries of the vector! If you just write

since auto skips all the quantifiers

```
for (auto i : a) i=10.56;
```

the i will contain a copy of a vector element! You are changing a copy, not the vector elements, the vector remains unchanged!

An important note

The iterators to a vector are (obviously) invalidated when memory is reallocated! So be careful with all operations that may reallocate memory, like push_back() or emplace_back().

```
// a vector with 8 doubles equal to 10.
std::vector<double> a(8,10.)
it=a.begin();
v=a[5];// OK v=10
a[2]=-7.6;// OK
a.push_back(7.8);// add new value
 //memory may have been reallocated
c=*it; //NO! it may be invalid!
```

const_iterator

A const_iterator (iterator to constant values) is an iterator that allows to access the elements read-only.

```
vector<float> a;
....
vector<float>::const_iterator b(a.cbegin());
*b=5.8;// ERROR!!!
```

Methods cbegin() and cend() are equivalent to begin() and end() but return iterators to constant values. You cannot change the element of the vector, only get them.

Going reverse

We can use special iterators to traverse a vector in a reverse order. Suppose we want to compute the convolution of two vectors $\sum_{i=0}^{n-1} a_i b_{n-1-i}$. Using iterators

```
using vect=std::vector<double>;// for convenience
double convol(vect const & a, vect const & b)
{
 auto j=b.crbegin()//const reverse iterator
 double res(0);
 for( auto const & v : a) res+=(*j++)*v;
}
```

*j++ "advances" iterator j returning the old value, which is dereferenced (dereferencing operator has lower precedence than the post-increment operator, see here). But, being j a reverse iterator, advancing ... means retreating! A little piece of obscure C++ programming :)

Interfacing with legacy code

Sometimes it is necessary to access the memory area of a vector<> through a pointer.

```
double myf(double const * x, int dim);//requires double*
...
vector<double> r;
...
y=myf(r.data(),r.size());
```

Note: You are not allowed to allocate/deallocate data using the pointer! Statements like r.data()=**new double**[100] are FORBIDDEN!. Memory handling of a std::vector should be made with the methods of the class.

Main methods of `vector<T>`

- Constructors `vector<T>()`, `vector<T>(int)` e `vector<T>(int,T const &)`
- Addressing (`[int]` e `at(int)`)
- Adding values: `push_back(T const &)` and `push_front(T const &)`
- Dimensions: `size()` e `capacity()`
- Memory management: `resize(int,T const &=T())`, `reset(int)` `shink_to_size(int)`
- Ranges `begin()` e `end()`
- Swap `swap(vector<T> &)`
- Clearing (without releasing memory): `clear()`
- Accessing data: `data()`

Main types defined by `vector<T>`

- `vector<T>::iterator` Iterator type
- `vector<T>::const_iterator` Iterator to constant values
- `vector<T>::value_type` The type of the stored elements (equal to `T`)
- `vector<T>::size_type` integral type used for indexes
- `vector<T>::pointer` (`vector<T>::const_pointer`) Pointer to elements (const variant)
- `vector<T>::reference` (`vector<T>::const_reference`) Reference to elements (const variant)

array<T,N>

std::array<T,N> is a container that encapsulates constant size arrays.

It is an extension of C-style array and has an interface quite similar to that of std::vector. The size and efficiency of array<T,N> is equivalent to size and efficiency of the corresponding C-style array T[N]. However, it provides the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc, and memory management (it obeys the RAII principle).

It provides most methods of a vector<T> a part those which involve dynamic memory management.

The second template argument is the size of the array

examples of standard arrays

```
std::array<double,5> a; //an array of 5 elements
std::array<double,6> b(4.4); //all elements initialised to 4.4
std::array<int,3> c{1,2,3};//aggregate initialization
std::array p{1,2,3};//automatic template ded. Array of 3 ints
c.size(); // dimension of array (3)
std::sort(std::begin(a), std::end(a));// Sorting the array
for(auto s: a) std::cout << s << ' '; //Range-based for
std::array<std::array<double,3>,3> m; // a simple 3x3 matix
m[2][0]=-7.8;// setting an element of m
```

Note that here I have used the free functions std::begin() and std::end(), but I could have used the equivalent function members of std::array.

Structured Bindings

std::arrays are aggregates (I will tell later on what it means). As a consequence you have at disposal use a special construct to extract their content, called structured binding, which may be very handy in several occasions, and you have aggregate initialization

```
std::array<double 2> fun()// a function returning an array
{ ... //compute a and b
 return {a,b}; // aggregate initialization
}
...
// now I use fun
auto [x,y]=fun(); // the elements of the array are in x and y
```

A note: in **auto** [x,y]=fun(); x and y are initialised with the corresponding array elements, i.e. they cannot be already existing variables.

An example of use of std::array is available in
Arrays/main_array.cpp

Testing the size of an array at compile time

The method size() for a std::array is a constexpr function, so it is computed compile-time, differently than the analogous method for std::vector. Indeed the size of a vector cannot be determined compile-time, since it may change run-time!
Therefore it may be used in a context where you need a constant expression!

```
template<unsigned int N>
class MyClass{...}; // a template class

#include <array>
int main(){
  std::array<float, 3> a;
  ...
  // I can use a.size() as template argument
  MyClass<arr.size()> m;
}
```

Tuples

Tuples are fixed size collections of object of different types.

```
#include <tuple>
...
using namespace std;
// Create a tuple.
tuple<string,int,int,complex<double>> t;
// create and initialize a tuple explicitly
tuple<int,float,string> t1{41,6.3,"nico"};
// use the utility make_tuple.
tuple<int,int,string> t2 = std::make_tuple(22,44,"nico");
// automatic deductiun (be careful)
tuple t3={3,4,5.0}; /a tuple<int,int,double>
```

See the examples in STL/tuple/test_tuple.cpp.

Suggestion: always use make_tuple to create a tuple.

Extracting/changing elements of a tuple

It is not possible to access a tuple with something as simple as the [] operator, because it contains elements of different type. You may

- Use the utility std::get<>

```
std::get<0>(t1)="a new string"; // change 1st element
int g = std::get<1>(t); // extract second element;
auto x = std::get<1>(t1); // of course you can use auto
```

- Tuple is an aggregate, so you can use structured bindings

```
auto [s,i,j,c] = t1;
auto & [k,l,x] = t3;
x=100.; // I am changing the third element of t3!
```

Extracting/changing elements of a tuple

- Use the utility std::tie to tie existing objects

```
// a function returning a tuple
std::tuple<int, double,double> fun();
...
int i; double a; double b;
std::tie(i,a,b)=fun();// tuple is unpacked into i, a and b
```

Note that you can ignore some elements of the tuple using the special object std::ignore.

```
std::tie(i,std::ignore,b)=fun();// tuple is unpacked into i and b
```

With tie you can also assign values to a tuple

```
std::tuple<int, double,double> t;
t= std::tie(i,a,b);// i a and b are copied in t
```

With structured bindings you create new variables, with tie you use existing ones.

pair

The header <utility> introduces pair<T1,T2> (loaded also by <map>), which is equivalent to a tuple with just 2 elements and in addition two members, called first and second

```
#include <utility>
...
std::pair<double,int> a{0.0,0};// Initialized by zero
// A very useful utility is make_pair
a=std::make_pair(4.5,2);
// first and second returns the values
auto c=a.first; //c is a double
int d=a.second;
```

Suggestion: always use make_pair to create a pair.

Vectors and matrices for numerics

The C++ language does not provide classes for matrices for scientific computing directly, even if the native data structure may form a valid base.

In this course we will use the Eigen vector and matrix classes, highly optimized for SSE 2/3/4, ARM and NEO processors.

We will make a special lecture on that.

Advanced Programming for Scientific Computing (PACS)
Lecture title: Smart pointers and references

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

RAII

Resource Acquisition Is Initialization is a big karma in C++. It basically means that an object should be responsible for the creation and destruction of the resources it owns. A terrible name, also the inventor admits it. Examples:

which is not true if we use classical ptrs

```
double * p =new double[10];
```

This is not RAII compliant! Who is destroying the resources pointed by p?? You have to take care of it!

```
std::array<double,10> p;
```

This is instead RAII compliant!. p is in charge of creating 10 doubles and of destoying them when it goes out-of-scope.

C++ smart pointers are another tool to implement RAII.

Pointers, smart and not

In modern C++ we use different types of pointers

- Standard pointers. Use them only to watch (and operate on) an object (resource) whose lifespan is independent from that of the pointer (but not shorter);
- Owning pointers. Also called smart pointers. They control the lifespan of the resource they point to. They are of 2 kinds:
  - unique_ptr, with unique ownership of the resource. The owned resource is destroyed when the pointer is destroyed (goes out of scope);
  - shared_ptr with shared ownership of a resource. The resource is destroyed when the last pointer owning it is destroyed.

Smart pointers implement the RAII concept. For just addressing a resource (maybe polymorphically) use ordinary pointers.

Smart pointers

We have:

unique_ptr<T>	Implements unique ownership. The resource is released (deleted) when the pointer goes out of scope
shared_prt<T>	Implements shared ownership. The resource is released when the last shared pointer goes out of scope
weak_prt<T>	A non-owning pointer to a shared resources. For special usage.

They all require the `<memory>` header.

a non-lock to check if the resource associated to a smart-ptr is still there or freed

The need of unique_ptr

Let's look at this example, where Polygon is the base class of several polygons:

```
class myClass{
setPolygon(Polygon * p){my_polygon=p;}
...
private:
Polygon * my_polygon;// Polymorphic object
};
// A Factory of Polygons
Polygon * polyFactory(std::string t){
switch(t){
case ''Triangle'': return new Triangle;
case ''Square'':      return new Square;
..
default: return nullprt;}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

"But where will be the delete?"

A poor design

This design is prone to errors.

First of all objects of type MyClass have now to take care of handling of the resource my_polygon (whenever you see a **new**, always ask yourself: "where is the **delete**?".)

We have to build constructors, destructor, assignment operator very carefully, and to account for possible exceptions to avoid memory leaks and dangling pointers.

There is always the risk that the user calls polyFactory and forgets to delete the returned pointer when it is required, causing a memory leak that may be very difficult to detect!

The version with unique_ptr

```
class myClass{
setPolygon(unique_ptr<Polygon> p){my_polygon=std::move(p);}
...
private:
unique_ptr<Polygon> my_polygon;
}
// A Factory of Polygons
unique_ptr<Polygon> polyFactory(std::string t){
switch(t){
case ``Triangle'': return std::make_unique<Triangle>();
case ``Square'':   return std::make_unique<Square>();
..
default: return unique_ptr<Polygon>();// null ptr
}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

Complete example in SmartPointers

How a unique_ptr<> works

so we cant copy them. But we can be moved (?)

A unique_ptr<T> serves as unique owner of the object (of type T) it refers to. The object is destroyed automatically when its unique_ptr gets destroyed.

It implements the * and the -> dereferencing operators, so it can be used as a normal pointer.

But it can be initialized to a pointer only through the constructor:

```
std::unique_ptr<int>up=new int;// ERROR!
std::unique_ptr<unt> down(new int);//OK!
```

or, much better, using the utility std::make_unique<T>()

```
auto p=std::make_unique<Triangle>();
```

The default constructor produces an empty (null) unique pointer. You can check if a unique_ptr is empty by testing if(prt.) or simply (but smart pointer are also contextually convertible to bool).

Moving a unique_ptr around

Unique pointers cannot be copied, for abvious reasons, but they can be moved (for details wait the lecture on move semantic).

Ownership can be transferred using the std::move utility:

```
unique_ptr<double> c(new double);
unique_ptr<double> b;
b= std::move(c);
```

b will point to what c pointed
c now will be set to nullptr

Now c is empty and b points to the double originally held by c.

Dealing with C-arrays

By default a unique_ptr calls **delete** for an object of which it loses ownership. Unfortunately, this will not work properly if the object is an array. However, there is a specialization that works for arrays:

```
unique_ptr<string> up(new string[10]);// A SERIOUS ERROR!!
unique_ptr<string[]> up(new string[10]);// OK
auto up=make_unique<string[]>(10);    // EVEN BETTER!
```

Suggestion: try to avoid C-style arrays. Use std::array, for which you do not need this specialization.

Main methods and utilities of unique_ptr

pt.empty() — to check if is a nullptr or has a real resource

swap(ptr1,ptr2)	Swaps ownership
pt1=std::move(pt2)	Moves resources from pt2 to pt1. The previous resource of pt1 is deleted. pt2 remains empty.
pt.reset()	Resource is deleted. pt is now empty. — like is a nullptr
pt.reset(pt2)	as pt=std::move(pt2)
pt.release()	returns a standard pointer. It releases the resource without deleting it. pt is now empty.

don't use it

unique_ptrs can be stored in a standard container:

```
vector<unique_ptr<Polygon>> polygons;
```

Shared pointers

so they need to track all the connections between them

While unique_ptr do not cause any computational overhead (they are just a light wrapper around an ordinary pointer), shared pointers do, so use them only if it is really necessary.

For instance you have several objects that "refer" to a resource (a Matrix, a Mesh...) that is build dynamically (and maybe is a polymorphic object). You want to keep track of all the references in such a way that when (and only when) the last one gets destroyed the resource is also destroyed.

To this purpose you need a shared_ptr<T>. It implements the semantic of "clean it up when the resource is no used anymore".

See the example in SmartPointers

The std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference (here reference is used in a generic sense) to an object that is managed by std::shared_ptr.

It must be converted to std::shared_ptr in order to access the referenced object.

It may be used to test if a resource associated to a shared_ptr has been deleted, in a thread-safe way.

However, it's usage is rather special and we omit the details here. You may find them in any good reference.

(l-value) references

References creates alias to existing objects. They must be initialized. Beware of reference to temporary objects!!

```
// A function returning a Matrix
Matrix hilbert(unsigned int);
double pippo(double const &);
...
double c= pippo(3);// OK
Matrix & a=horner(5)//NO;
Matrix const & c=horner(3);OK!
double * pz= new double;
double & z=*pz;//OK, so far
z=5.0;// OK *pz is now 5
delete pz;// NO z is now ``dangling''
z=7;//a segmentation fault is granted!
```

the return value of a funct can be bounded only to a const ref

in this case we will store that object (no so like an exception)

as the object to which refers does not exist anymore

A const reference prolongs the life of a temporary object.

The use of references

References are very important in C++ programming and essential in C++ for scientific computing. Their main usage is as parameter (and sometimes return type) of a function. Passing a reference instead of a value help saving memory, since you refer to an existing object, and also (if the reference is not const) it provides an alternative way to return data from the function.

The initialization of a reference is called binding, and the binding rules are an essential feature of references, particularly when coupled with function overloading. We will postpone the discussion in the lecture on move semantic, where we also introduce a new beast: the r-value reference.

Reference semantic in std containers

Std containers can hold only "first class" objects, but not references (they are not first class objects, but alias to already existing ones):

```
vector<unique_ptr<Polygon>> a; //OK
vector<Polygon*> a; //OK
vector<Polygon &> n; //ERROR!
```

Reference semantic in std containers

However, we have a way to store obiects with the same semantics as references in a container. You need to use std::reference_wrapper defined in the header <functional>

```
Point p1(3,4);
Point p2(5,6);
std::vector<std::reference_wrapper<Point>> v;
v.push_back(p1);
v.push_back(p2);
v[0].setCoord(7,8);// changes coordinates of p1
```

A const reference prolongs the life of a temporary object

```
Matrix pippo()
{
  // An object in the scope of the function
  Matrix m=identity(10,10);
  return m;
}
main()
{
  Matrix const & a=pippo();
  ....
}
```

The lifespan of the (temporary) Matrix returned by pippo() is extended to be the same as that of the constant reference a
So you can safely use a in your program.

References vs pointers

There is often confusion on reference and pointers. Let's try to make things clear

- A pointer is a variable, it uses memory (normally 8 bytes) to store a memory address. When dereferenced returns the value associated to the address it points to (provided it is a valid address). But for the rest, it is like any other "first class object": it can be reassigned, changed etc.;
- You can have null pointers, i.e. pointers that points to nothing, which you can assign to the address of an object later on;
- A pointer type is indeed a different "base type" than that of the pointed object. A `double*` is a complete different type than `double`.
- A reference is just an alias to an existing object. A reference to a variable gives a "secondary name" to that variable. You cannot have unbound references, nor you can bind a reference to another object. Reference and referenced object are bound for life!
- A reference does not define a different "basic type", sometimes it is said that it is an adornment to a type. A `double&` behaves exactly as a `double`.

What about r-value references?

We will not discuss in this course on rvalue references, the one with two amperstands (like **double**&&). I will add to WeBeep some information for the one of you who wants to know more about move semantic, rvalues etc..

Advanced Programming for Scientific Computing (PACS)
Lecture title: Functions, functors and lambdas

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Functions in C++

In C++ functions may be free functions or member functions (also called methods) of a class. And a further subdivision can be made between normal functions and template functions (including automatic return functions).

For a free, non template function the usual declaration syntax is

```
ReturnType FunName(ParamsType...);
```

or

```
auto FunName(ParamsType...)-> ReturnType;
```

or (automatic return function)

```
auto FunName(ParamsType...);
```

alternative ways to define a function

The first two forms are practically equivalent. So choose the one you like most.

In the third possibility, the return type is deduced by the compiler, we will give more details later on.

Why (free) functions?

A function represents a map from data given in input (through the arguments) and an output provided by the returned value or possibly via an argument if the corresponding parameter is a non-const reference.

Consequently, a (free) function is usually (I say usually because there are exceptions) stateless, which implies that two different calls of a function with the same input produce the same output.

Therefore, you normally implement a function whenever what you need is indeed a map input/output, like in the standard mathematical definition $f : U \to V$.

void return type and [[nodiscard]]

The return type can be **void**, which means that the function is not returning anything. Indeed **void** is a type that indicates "no value".

In C++ it is not an error not using the returned value: this is a perfectly valid piece of code

```
double fun(double x);
...
fun(4.57);// OK, returned values is discarded.
```

If you want to get a warning if the value is discarded you should use the `[[nodiscard]]` attribute

```
[[nodiscard]] double fun(double x);
...
fun(4.57);// Warning issued
```

You may also indicate a string, printed in case of discarded value:

```
[[nodiscard("Strategic value")]] double fun(double x);
```

Function declaration and definition

A function pure declaration does not contain the body of the function, and is usually contained in a header file:

```
double f(double const &);
```

It is not necessary to give a name to the parameters (but you can if you want, and it is good for documentation).

The definition is usually contained in a source file, a part from (non fully specialized) function templates, automatic return functions, inline and contexpr functions, which should be defined in a header file.

```
double f(double const & x)
{
  //... do someting
  return y;// returns a value convertible to double
}
```

Passing by value and passing by reference

I assume you already know what is meant by passing a function argument "by value" or "by reference" . In fact, "passing by reference" simply means that the function parameter is a reference, the function operates on an "alias" of the argument, without making a copy:

- Using a non-const reference, a change made in the function changes the corresponding argument, the parameter can be considered as a possible output of the function. The argument cannot be a constant object or a constant expression;
- Using a **const** reference, the parameter is a read-only alias, so the argument can also be a constant object or a constant expression. The parameter cannot be changed in the function.

Important Note: In this lecture with "reference" we usually mean l-value references. When we will introduce move semantic, we will describe r-value references (&&).

Passing a value

If you pass an argument "by value", the value is copied (or possibly moved, as we will see in a later lecture) in the parameter, which is then a local variable of the function: a change in the parameter value does not reflect on the corresponding argument.

When you pass a value, the only reason you may want the parameter to be declared **const** is to have a safer code and be sure that any attempt of changing the parameter inside the function, even indirectly by passing it by reference to another function, is forbidden.

A Note: Another possibility of obtaining the output of function call via the arguments is by using pointers. This is the only technique available in C. In C++ references are preferred for this purpose.

A basic example

```
double fun (double x, double const & y, double & z)
{
 x = x +3; // I am changing a copy of the argument
 auto w= y; // I am copying the value bound to y into w, ok
 y = 6.0; // ERROR I cannot change y
 z = 3*y; // Ok the variable bound to z will change
 ....}
 ...
 double k=3.0;
 double z=9.0;
 auto c = fun(z, 6.0,k); //ok
 // k is now 18 z is still 9.0
 auto d = fun(z,k,6.0); //Error 6.0 cannot bind to a double &!
```

General guidelines I

- Prefer passing by reference when dealing with large objects. You avoid to copy the data in the function parameter and save memory.
- If the parameter is not changed by the function (it is an "input-only parameter"), either pass it by value (better if **const**) or as const reference (es: **const** Vector &).
- A temporary object or a constant expression may be given as argument only if passed by value or by constant reference (or...let's wait the lecture on move semantic)
- You can pass polymorphic objects only by reference or with pointers. Otherwise, polymorphism is lost!

General guidelines II

- In the "passing by value" case, i.e. when the parameters are not references, the use of **const** is less critical, since we are using in the function a copy of the argument. However, it does not hurt declaring **const** what is not changed inside the function (and helps the compiler to produce better machine code).
- In case of POD (**int**, **double**) there is little gain in passing by const reference. Particularly with **int**, since the compiler may decide to store the parameter in a CPU register. But it is just a matter of taste.

In Bindings/ you find an example about reference binding (look only at the part concerning l-value references).

What type does a function return?

A free function normally returns a value or **void** (no value returned).

Never return a non-const l-value reference, or pointer, to a local function object

without this would have worked. With it we get an error as we can't return a ref to m, since m is a local variable which will vanish when we return

```
Matrix & pippo()
{
    // An object in the scope of the function
    Matrix m=identity(10,10);
    return m;
}
```

This is wrong, even if you change it in Matrix **const** & pippo().

Always return by value objects created in the scope of a function

What type does a function return?

In some rare cases a function returns a l-value reference to an object passed by l-value reference. Normally when you want concatenation. A typical case is the streaming operatpr

```
std::ostream & operator<<(std::ostream & out, Myclass const & m)
 {
    ....// some stuff
    return out;// return the stream
 }
```

This allows concatenation:

```
std::cout<<myclass<<" concatenated with  "<<x;
```

What type does a function return?

In the case of a method of a class, you may return a reference to a member variable, to allow its modification

```
double & setX(){return this->x_;} // x_ variable member
...
x.setX()=10;// I change the member
```

or, occasionally, const references when you want to have the possibility of using potentially big variable members without copying them.

```
Matrix const & getM()const {return this->bigMatrix_;}
...
y=x.getM()(8,9);// get an element of the matrix
```

A typical case is the subscript operator, **operator**[](**int**);
We will see more examples in the lecture about classes.

inline

The **inline** attribute applied to a function declaration used to suggest the compiler to "inline" the function in the executable code, instead of inserting a jump to the function object.

In modern C++, **inline** simply means: "do not apply the one-definition rule" to this function (or variable). Definitions of **inline** functions are in header files. They will be recompiled in all translation units that use the functions, but the linker will not complain about multiple definitions, it will just use the first one.

Still, compilers, if optimization is activated, are free to "inline" an inline function if they deem it appropriate (you can avoid it with a special attribute, but we omit giving more detail).

An example of **inline**

```
inline double cube( double const & x)
{
   return x*x*x;
}
```

It should be placed in a header file.

Implicit **inline** functions

You do not need to write **inline** (it is implicit) in the case of

- Function templates (unless fully specialized or explicitly instantiated)
- Methods defined "in-class"
- `constexpr` functions
- Automatic parameter functions

They should be all defined in a header file.

You do not need to, but if you write **inline** in those cases it is not an error. Just redundant.

constexpr functions

With the **constexpr** specifier we are telling the compiler to try computing the function at compile time whenever possible.

```
// the argument is given by value
constexpr double cube(double x)
{ return x*x*x;}
```

By doing so the compiler may evaluate at compile time the expression

```
a=cube(3.0); // replaces it with a=9.0
```

A `constexpr` function has to comply with some restrictions, the main ones:

- It can call only other constexpr functions;
- Cannot be a virtual method;

.

constexpr functions

- Constexpr functions should be defined in a header file
- Constexpr implies inline: if an argument of the function is not a constant expression, a constexpr function behaves like a normal inline function;
- Constexpr functions make sense only for (A) small functions that you will call many times in your program (this is true also for inline functions); (B) when the value returned by the function is used in a context where you need a constant expression. Take for example the method size() of an array: if it had not been **constexpr** you could not use it as template value argument.

Otherwise, write a normal function.

Examples of implicit **inline** functions

```
// function template
template< class T>
 T fun(const T & x){ ..... }

 class MyClass{
 ...
 // in class definition
 double fun(double x){ return 5*x;}
 }
 // constexpr function
 constexpr double f(double x){return 1./x;}

 // Automatic parameter function (C++20)
 double fun(auto const& x, auto y){...}
```

Recursive functions

A function can call itself

```
double myPow( double const & x,unsigned n)
{
if(n==0) return 1.0;
else
return x*myPow(x,n-1);
}
```

We will see other examples with template functions.

A suggestion: recursive (standard) functions are elegant but non necessarily efficient. Often, non recursive implementations are better.

Default parameters

In the declaration of a function, the rightmost parameters may be given a default value.

```
vector<double> crossProd(vector<double>const &,
vector<double>const &, const int ndim=2);
...
a=crossProd(c,d);//it sets ndim=2
...
```

Default constructed arguments (another use of braces)

There are cases where one wants to pass as argument a default constructed argument, or, in general an object constructed on the fly. You can enjoy another goody of brace initialization.

```
double f(std::vector<double> const & v, double x, int k);
....

// passing an empty vector
 y = f({}, 4.,6);
// passing a vector of three doubles
 z = f({1.,2.,3.}, 5. ,6);
...
 u = f(v,3.,{});// Same as f(v,3,0)
```

The parameter must be either a value or a const- lvalue reference (or a r-value reference)

Static function variables

In the body of a function we can use the keyword **static** to declare a variable whose lifetime spans beyond that of the function call.
A static variable in a function is visible only inside the function, but its lifespan is global. They are useful when there are actions which should be carried out only the first time the function is called.

```
int funct(){
  static bool first=true;
  if(first){
      // Executed only the first time
        first=false;}
  else{
      //Executed from the second call onwards
      ...
      }
```

In this case the function is not stateless anymore!

Pointers to functions

```
double integrand(double x);
...
using Pf = double (*)(double)
//typedef double (*Pf)(double);
double simpson(double, double, Pf const f, unsigned n);
...
// passing function as a pointer
auto integral= simpson(0,3.1415, integrand,150);
// Using a pointer to function
Pf p_int=std::sin;
integral= simpson(0,3.1415, P_int,150);
...
```

The name of the function is interpreted as pointer to that function, you may however precede it by &: Pf p_int=&integrand;.

We will see in a while a safer and more general alternative to function pointers, the function wrapper of the STL.

An example

In the directory Horner you find an example that uses functions and function pointers to compare the efficiency of evaluating a polynomial at point x with two different rules:

- Classic rule $y = \sum_{i=0}^{n} a_i x^i$;
- The more efficient Horner's rule:

$$y = (\ldots(a_n x + a_{n-1})x + a_{n-2})x \ldots + a_0$$

Try with an high order polynomial (e.g. $n = 30$) and try to switch on/off compiler optimization acting on the local `Makefile.inc` file, and even activate parallelism using a standard algorithm!

Function names and function identifiers

A function is identified by

- Its name, fun in the previous examples. More precisely, its full qualified name, which includes the namespace, for instance std::transform.
- The number and type of its parameters;
- The presence of the **const** qualifier (for methods);
- The type of the enclosing class (for methods).

Two functions with different identifiers are eventually treated as different functions. It is the key for function overloading.

Note: the return type is NOT part of the function identifier!

A recall of function overloading

```
int fun(int i);
double fun(double const & z);
//double fun(double y);//ERROR! Ambiguous
 ....
 auto x = fun(1);// calls fun(int), x is a int
 auto y = fun(1.0);// calls fun(double const &), y is a double
```

The function that gives the best match of the arguments type is chosen. Beware of possible ambiguities, and implicit conversions!

(like a double and a (const) reference to a double, one implicit conversion

function templates

We will discuss templates in details a special lecture. Yet, we can anticipate function templates, since their use is quite intuitive.
A function template is not a function: it is the template of a function, where some types are parametrised and unknown when writing the template. Only at the moment of the instance, i.e. when the function is used, the parameter type can be resolved and the compiler can produce and compile the corresponding compiled code.
Function templates are useful when you want a function that may work for several argument types and you want to spare avoid repetition.

What is nice in function templates is that the parameter type may be deduced from the type of the arguments given to the function.

function templates

T will become like an alias for the deduced type of the parameter

we write a function f in this way

```
template<class T>
double f(T const & x){...}
```

The template function parameters are automatically deduced from the type of the corresponding argument when the template is instantiated. For instance,

then the compiler tries to deduce the type for T from how we call that f

```
auto y=f(5.0);
```

will instantiate a **double** f(**double const** &), that is the function obtained by setting T=**double**.

It is possible to fully specialize a template function to bound it to specific argument types. Here, an example of full specialization for **double**:

if we call this f, then this specialization gets the priority over the generic-template ones

```
template<>
double f(const std::complex<double> & x){...}
```

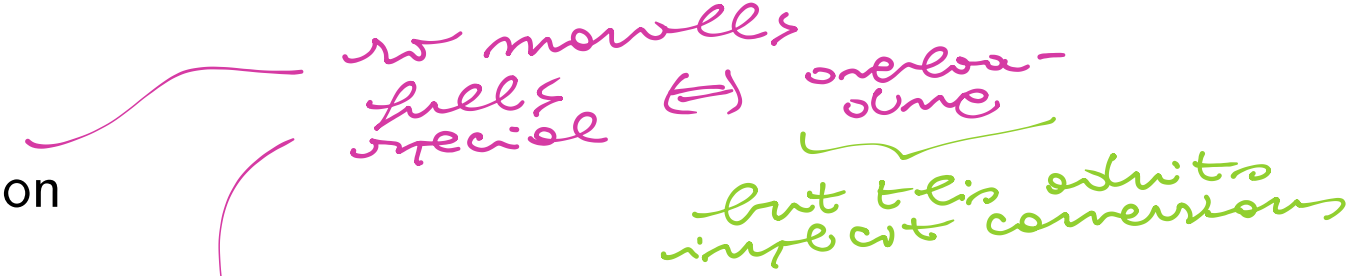
Now x=f(std::complex<**double**>{5.0,3.0}) will use the specialized version.

Overloading for special cases (it is not a specialization)

```
// Primary template
template <class T>
T dot(std::vector<T> const & a, std::vector<T> const & b)
{
        T res =0;
        for (std::size_t i= i; i<a.size();++i) res+=a[i]*b[i];
        return res;
}
        // overloading for complex
template<class T>
T dot(std::vector<std::complex<T>> const & a,
     std::vector<std::complex<T>> const & b)
{
        T res =0;
        for (std::size_t i= i; i<a.size();++i)
                res+=a[i].real()*b[i].real()
                    +a[i].imag()*b[i].imag()_;
        return res;
}
```

Now dot(x,y) calls the version for vector of complex numbers if a and b are complex.

A note



The full specialization

```
template<>
double f(const std::complex<double> & x){...}
```

can be replaced by an overloading, just omitting the **template** construct.

```
double f(const std::complex<double> & x){...}
```

The result is almost the same. The main difference is that with full specialization implicit conversion is not admitted, the type must be matched exactly.

Recursion with function templates

Template parameters can also be integral constants, this allows compile time recursion:

```
        // Primary template
template<unsigned N>
constexpr double myPow(cost double & x){ return x*myPow<N-1>(x);}
// Specialization for 0
template<>
constexpr double myPow<0>(const double& x){return 1.0;}
...
double y = myPow<5>(20.); // 5^20
```

now we have a normal function but with another "parameter" N (known at compile time) ⇒ more efficient

this call (thanks also to constexpr) will be solved at compile time

The following version is even better:

```
template<unsigned N>
constexpr double myPow(const double & x)
 if constexpr (N==0) return 1.0;
 else                return x*myPow<N-1>(x);
 }
```

needed for compile-time resolution

An important note

A thing to remember: implicit conversion does not apply to template function arguments

```
template<class T>
 double fun(T a, T b);
 ...
 int i=9;
 double x=20.;
 // ERROR compiler cannot resolve fun(int, double)
 double y=fun(i,x);
 // Ok I am explicitely converting int->double
 double y =fun(static_cast<double>(i),x)
 // Or I can explicitely tell which version I want
  double y =fun<double>(i,x)
```

Note: of course I may do

```
template<class T1, class T2>
double fun(T1 a, T2 b);
```

Automatic parameter functions

We have a simpler (yet less flexible) way to construct function taking arbitrary argument types. We can replace

```
template<typename T, typename M>
double fun(T const & x, M y);
```

with

```
double fun(auto const & x, auto y);
```

Simpler, isn't it? (why have they waited until C++20?).

Automatic parameter functions are just function templates in disguise, so their definition is in a header file.

Automatic return type deduction

We can let the compiler deduce the return type, both for ordinary functions and templates, even if the technique is more useful for templates (or automatic functions)

```
auto mul(auto const & x , auto const & y){
return x*y;
}
```

we have to add these (if we want) so auto gets just the basic type

The return type is deduced by the compiler. For instance,

```
double x=9.56;
int j =9;
...
auto x = mul(x,j);// x is a double
auto y = mul(1,j);// y is an int
// here z is a std::complex<double>
auto z = mul(x,std::complex<double>{1.,2.});
```

decltype(**auto**)

this extracts the whole type

Remember that **auto** strips qualifiers and references. So is you want a function return an automatically deduced reference (it makes sense only for methods) you have to do

```
auto & iAmReturningARef();
```

In special situations you may need decltype(**auto**) instead of **auto**: typically when you want an adaptor to forward the result of the call to another function.

a thing to control the interface of a function, like a wrapper

```
decltype(auto) adapter(const double & x)
{
    return aFunction(x,4,0);
}
```

return "what should be returned" from the function

Maybe I do not know what type aFunction returns. But here I do not care, I just grab the exact type of the returned value.

Functors (function object)

A function object or functor is a class object (often a struct) which overloads the call operator (`operator()`). It has a semantic very similar to that of a function:

```
struct Cube{
  double m=1.0;
  double operator()(double const & x) const {return m*x*x*x;}
};
...
Cube cube{3.}; // a function object, cube.m=3
auto y= cube(3.4)// calls operator()(3.4)
cube.m=9; // change cube.m
auto l= cube(3.4)// Again with a different m
auto z= Cube{}(8.0)//I create the functor on the fly
```

If the call operator returns a `bool` the function object is a predicate. If a call to the call operator does not change the data members of the object you should declare **operator**() `const` (as with any other method).

Why functors?

A characteritic of a functor is that it may have a state, so it can store additional information to be used to calculate the result

```
class StiffMatrix{
public:
 StiffMatrix(Mesh const & m, double vis=1.0):
    mesh_{m},visc_{vis}{}
 Matrix operator()() const;
private:
 Mesh const & mesh_;
 double visc_;
};
...
StiffMatrix K{myMesh,4.0}; //function object
...
Matrix A=K();// compute stiffness
```

STL predefined function objects

Under the header <functional> you find a lot of predefined functors

```
vector<int> i={1,2,3,4,5};
vector<int> j;
transform (i.begin(),i.end(),   // source
           back_inserter(j),      // destination
           negate<int>());
```

Now j={−1,−2,−3,−4,−5}. Here, negate<T> is a *unary functor* provided by the standard library.

Note: I need a back_inserter as I am inserting the transformed elements at the end (back) of vector j.

Some predefined functors

plus<T>	Addition (Binary)
minus<T>	Subtraction (Binary)
multiplies<T>	Multiplication (Binary)
divides<T>	Division (Binary)
modulus<T>	Modulus (Unary)
negate<T>	Negative (Unary)
equal_to<T>	equality comparison (Binary)
not_equal_to<T>	non-equality comparison (Binary)
greater, less ,greater_equal, less_equal	
logical_and<T>	Logical AND (Binary)
logical_or<T>	Logical OR (Binary)
logical_not<T>	Logical NOT (Binary)

For a full list have a look at this web page.

Lambda expressions

We have a very powerful syntax to create short (and inlined) functions quickly: the lambda expressions (also called lambda functions or simply lamdas). They are similar to Matlab anonymous functions, like f = @(x) x^2. Let's look at a simple example.

```
...
auto f= [](double x){return 3*x;};// f is a lambda function
...
auto y=f(9.0); // y is equal to 27.0
```

Note that I did not need to specify the return type in this case, the compiler deduces it as decltype(3*x), which returns **double**.

Lambda syntax

The definition of a lambda function is introduced by the [], also called capture specification, the reason will be clear in a moment. We have diffferent possible syntax (simplified version)

```
[ capture spec]( parameters){ code;  return something}
```

or

```
[ capture spec]( parameters)-> returntype
{ code }
```

The second syntax is compulsory when the return type cannot be deduced automatically.

The capture specification allows you to use variables in the enclosing scope inside the lambda, either by value (a local copy is made) or by reference.

like a way to include more arguments into the expression

[]	Captures nothing
[&]	Captures all variables by reference
[=]	Captures all variables by making a copy
[=, &foo]	Captures any referenced variable by making a copy, but capture variable `foo` by reference
[bar]	Captures only `bar` by making a copy
[**this**]	Captures the `this` pointer of the enclosing class object
[∗**this**]	Captures a copy of the enclosing class object

An example

The capture specification gives a great flexibility to the lambdas
We make some examples: return the first element i such that $i > x$ and $i < y$

```
#include<algorithm>
int f(vector<int> const &v,int x, int y){
auto pos = find_if (v.begin(), v.end(),   // range
     [x,y](int i) {return i > x && i < y;});// criterion
return *pos;
}
```

I have used the `find_if` algorithm which takes as third parameter a predicate and returns the iterator to the first element that satisfies it.

something that returns a bool

Other examples

```
  std::vector<double>a={3.4,5.6,6.7};
  std::vector<double>b;
  auto f=[&b](double c){b.emplace_back(c/2.0);};
  auto d=[](double c){std::cout<<c<<" ";};
  for (auto i: a)f(i); // fills b
  for (auto i: b)d(i); //prints b
// b contains a/2.

// We also have generic lambdas
auto fun2=[](auto const & x){return x+pi;};

auto x = fun2(3.0L); // x=6.1415 is a long double
```

Generic Lambdas

You can allow lambda functions to derive the parameter type from the type of the arguments. Another example...addition of std::**string** means concatenation.

```
auto add=[](auto x, auto y){return x + y;};
double a(5), b(6);
string s1("Hello ");
string s2("World");
auto c=add(a,b); //c is a double equal to 11
auto s3=add(s1,s2);// s is "Hello World"
```

Some notes:

1) avoid capturing all variables. It is better to specify just those you need.

2) **auto** is a nice feature. Don't abuse it. If the type you want is well defined, specifying it may help understanding your code.

Generalised capture

You can give alternative names to captured objects. Normally it is not needed, but it can be handy sometimes

```
double x=4.0;
...
// x captured by reference in r
// i is taken equal to x+1
auto f = [&r=x, i=x+1.]{
          r=r*4;
          return r*i;
          }
...
double res = f();
// Now x=16 and res=80.!
```

Multiple returns

Lambdas can have more then one return (like an ordinary function), but they must return the same type (no type conversion allowed on the returned objects).

```
 double f(){...} // ordinary fun
 auto g = [=] ()
{
 while( something() ) {
   if( expr ) {
       return f() * 42.;
    }
 }
 return 0.0;// & multiple returns
}           //   (types must be the same)
```

Lambdas as adapters (binders)

Lambda expressions may be conveniently used as binders: adapters created by binding to a fixed value some arguments of a function.

```
// A function with 3 arguments
double foo(double x, double y, int n);
// A function for numerical integration
template<class F>
Simpson(F f, double a, double b)...
// I want to do the integral of foo with respect
// to the first argument, the others given
auto f=[](double x){return foo(x,3.14,2);}
double r = Simpson(f,0.,1.);
```

You have in C++ also specific tools for "binding", but lambdas are of simpler usage.

lambdas and constexpr

constexpr variables are imported into the scope of a lambda expression with no need of capture:

```
constexpr double pi=3.1415926535897;
auto shiftsin=[](double const & x){return std::sin(x+pi/4.);};
...
double y = shiftsin(3.2);// y=sin(3.2+pi/4.)
```

This is nice feature of constant expressions.

An example of use of [this]

With **this** we get the **this** pointer to the calling object!

```
class Foo{
public:..
void compute() const;
private:
double x_=1.0;
vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[this](double a){x_*=a;};
    std::for_each(v_.begin(),v_.end(),prod);}
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Here compute() uses the lambda prod that changes the member x_.
To be more explicit you can write **this**->x_*=a;.

An example of use of [*this]

With *this we get a copy of the calling object! Here generic capture is handy:

```
class Foo{
public:..
void compute() const;
private:
double x_=1.0;
vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[foo=*this](double a){return foo.x*a};
    std::for_each(v_.begin(),v_.end(),prod);}
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Now, in the execution of the last statement, foo inside the lambda is a copy of myFoo.

Functions (or lambdas) returning lambdas

Here a template lambda function that returns a lambda function computing the approximation of the N-th derivative of a function by forward finite differences (see Derivatives/Derivatives.hpp for other solutions);

```
template <std::size_t N>
auto numDeriv=[](auto f, double h){
  if constexp (N==0u)
      return [f] (auto x){return f(x);}
  else{
      auto prev=numDeriv<N-1u>(f,h);
      return [=](auto x){
            return (prev(x+h)-prev(h))/h;};
  };
};
..
auto f=[](double x){x*std::sin(x);}
auto ddf=numDeriv<2>(f,0.001);
double x = ddf(3.0); //numerical II derivative at x=3.0
```

in both cases we return a function

Callable objects

The term callable object (or just callable) indicates any object f where the syntax f(args..) is allowed and may return a value. It can be

- An ordinary function;
- A pointer to function;
- A lambda expression;
- A function object (functor).

A callable object may be called in the usual way of by using the std::invoke utility present in the header <functional>. I am not giving more details on invoke, you may find them here.

Function wrappers

An now the catch all function wrapper. The class `std::function<>` declared in `<functional>` provides polymorphic wrappers that generalize the notion of function pointer. It allows you to use any callable object as first class objects.

```
int func(int, int); // a function
struct F2{ // a function object
int operator()(int, int) const;};
...
// a vector of functions
std::vector<std::function<int(int,int)>> tasks;
tasks.push_back(func);// wraps a function
tasks.push_back(F2{});// wraps a functor
tasks.push_back([](int x,int y){return x*y;});// a lambda
for (auto i : tasks) cout<<i(3,4)<<endl;
```

It prints the result of `func(3,4)`, `F2(3,4)` and 12 (3×4).

Function wrappers

Function wrappers are very useful when you want to have a common interface to callable objects.

See the examples in , RKFSolver, FixedPointSolver and NewtonSolver. The first implements a RK45 adaptive algorithm for integration of ODEs and systems of ODEs. The other two codes deal with the solution of non-linear systems.

Function wrappers introduce a little overhead, since the callable object is stored internally as a pointer, but they are extremely flexible, and often the overhead is negligible.

Stateless function objects

A function object is said to be stateless if its state does not depend from previous calls. Consequently, the object returns the same values when called twice with the same arguments.
For instance,

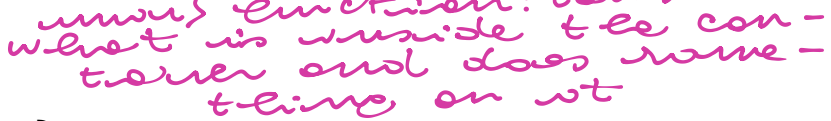
```
struct Count7{
    void operator ()(int i){if (i==7) ++count7;}
    int count7=0;
}
```

is not stateless (i.e it is stateful): it modifies the state of the Count7 object.

Be careful when passing stateful callable objects to algorithms of the standard library. Some care must be taken to avoid having bad surprises!

Stateless functors and std containers and algorithms

The concept of stateless functor is important because the standard library implicitly assumes the functors are stateless. Let's consider this example




```
int main(){
    std::vector<int> v={1,7,7,8,7,9};
    Count7 c7; // a counter of 7
    std::for_each(v.begin(),v.end(),c7); // apply c7 to v
    std::cout<<" The number of sevens is "<<c7.count7;
}
```

The answer is: `The number of sevens is 0`!!!! Why!!!

Analysis of the problem

Looking at cppreference.com I see that the declaration of std::for_each is

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each(InputIt first,InputIt last,UnaryFunction f);
```

It is a template function that takes as first two arguments two iterators and as third argument a UnaryFunction, that is a callable object that takes just one argument. So it looks all right but... the callable object is taken by value! It means that for_each makes an internal copy of c7, an uses it to operate on the vector. My c7 object is unchanged!! So the counter c7.count7 is still zero.

Is there a solution? Yes, actually two solutions, the first specific, the second more general.

The solution

A first solution, valid for for_each, is to note that it returns the callable object passed as argument!. So, you can do

```
c7 = std::for_each(v.begin(),v(end),c7);
```

to overwrite c7 with the one used inside for_each.

A second solution is using the magic of std::ref(), and do

```
std::for_each(v.begin(),v(end),std::ref(c7));
```

Now, c7 is passed by reference, so any operation inside for_each is reflected in my c7 object.

ref() is a small utility that returns a reference wrapper, a first-class object holding a reference to the argument. A little trick that may be used in all similar situations. The trick is applicable only on template function parameters.

For the nerds

How does std::ref work? Well, here my "poor man" version,

```
template <class T>
T& ref(T& x)
{
    return x;
}
```

Have you understood how it works? Think about it.

Another solution (actually my preferred one)

Use a lambda expression (they are explained later!)

```
Count7 c7;
std::for_each(v.begin(),v(end),[&c7](int i){c7(i);} );
```

Here c7 is captured as a reference by the lambda. The lambda will be copied, but the copy will just carry on the reference. So inside for_each I am using an alias of the c7 object in the calling code.

So, stateful functors can be perfectly fine, but you have to take some care. In particular, watch out for possible unwanted copies!

Side-effect

Another important concept is that of side-effect. A function has a side-effect if it modifies an object outside the scope of the function. So stateful callable objects have side-effect.

In this case, you need to be careful in a parallel implementation!

Let's consider our Cont7 example. In a distributed memory paradigm like MPI, each process has its own copy of a c7, so it counts the 7 in the portion of the vector assigned to it. To have the total number you have to sum them up!

If you use a shared memory paradigm instead, like threads or openMP, you must ensure that different threads do not access the counter at the same time! You have to make the counter update atomic!

Function Expression Parsers

Functions in C++ must be defined at compile time. They can be pre-compiled, put into a library file and even *loaded dynamically* (as we will see in a next lecture), Yet, you have still to compile them!.

Sometimes however it can be useful to be able to specify simple functions run-time, maybe reading them from a file. In other words, to interpret a mathematical expression, instead of compiling it.

This, of course introduces some overhead (after all we are using a compiled language for efficiency!). Yet, in several cases we can afford the price for the benefit of a greater flexibility!.

Possible Parsers

We need to use some external tools that parses a mathematical expression, that can contain variables, and evaluate it for a given value of the variables.
Possible alternatives (not exhaustive)

- Interface the code with an interpreter, for instance interfacing with Octave;
- Use a specialized parser. Possible alternatives: boost::spirit, μParser and μParserX, a more advanced (but slower) version of μParser.

μparser

In this course we will see μParser and μParserX.

A copy of the software is available in the directories Extras/muparser and Extras/muparserX. To compile and install it (under the Examples/lib and Examples/include directories) just launch the script indicated in the `README.md` file.

Those directories are in fact submodules that link to a fork of the original code, adapted for the course. That's one of the reasons of the need of `--recursive` when you clone the repository of Examples and Exercises.

You find a simple example on the use of μParser in mParserInterface/test_Muparser.cpp and the other files in that directory.