Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Introduction

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

General Information

The course consists of lectures, exercise sessions in computer room and a project valid for the final evaluation;
Students following the 8 credit version of the course may replace the project with an examination.

Lectures are held on
Wednesday from 10.15 to 12.00 room B5.2 and Thursday from 15.15 to 18.00 in room B5.3.
The exercise sessions are held on Friday from 8.15 to 11.00 in the Laboratorio Didattico of the Departement of Mathematics (fourth floor).

Lecturers:

- Lecturer: Luca Formaggia (luca.formaggia@polimi.it)
- Assistant: Carlo deFalco (carlo.defalco@polimi.it)
- Tutor Stefano Zonca (stefano.zonca@polimi.it)

Reception Hours: Wednesday 14.15 16.15 (on appointment)

Books and material

C++ and Object Oriented Numeric Computing for Scientists and Engineers, Daoqi Yang, Springer-Verlag, 2000
C++ Primer (5th edition), S. Lippman et. al, Addison-Wesley, 2012
Course Slides and Notes

Exams

8 credit version

- Homework exercises (3 points max)
- Final Test or (optionally) a project.

10 credit

- Homework exercises (3 points max)
- Project

On line resources

- The page of Couress on line Politecnico (BEEP) will be used as exchange point, in particular for handing in homework exercises. I will put ther a copy of the slides and other material.
- A *github* site has been set up for the course Examples and Exercices.

Virtual Machine

You will have access to the unix system of the Department of Mathematics. You will be able to access the system remotely (online) and will also be provided with a virtual appliance reproducing the same environment for off-line use.
To use the virtual machine you have to install VirtualBox. If you already use LinuX (or MacOs) you may avoid the Virtual Machine, but you will need to install the packages we use during the course by yourself!
If you already use LinuX (or MacOs) you may avoid the Virtual Machine, but you will need to install the packages we use during the course by yourself! And it will be up to you to check that your excercise/project runs correctly on the target system.

Examples

The example shown during the course are available from the github site https://github.com/pacs-course/pacs. You will find different directories

- `Exercises` The exercises for the course.
- `Extra` Some exte material: software, notes etc.
- `Examples` The examples.

In all directories a `README` files contains the description of the content.

The Examples

The directory `Examples` consists of several subdirectories:

- `incude`, where the include files used by more than one examples are stored;
- `lib`, where libearies 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.

To download the Examples (and the other material) do (<name> is a name of your choice).

```
mkdir<name>
cd <name>
git clone git@github.com:pacs-course/pacs.git
cd  Examples
cp Makefile.user Makefile.inc
```

The user has to change some variables in the Makefile.inc in the `Example` directory to suit his/her system.

To update the content (do it frequently!).

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

If you find a bug, or improve an example, you may submit the changes via the GitHub site!

Why the github site?

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 (and github) may be used also for the project for your exam!.

How to compile the examples

Copy `Makefile.user` in `Makefile.inc` and change `PACS_ROOT` to the directory where the Examples reside (the same directory of `Makefile.user`.
Go to the `src/Utilities`. Type `make`, check that the compilation is fine, and then `make install`.
To compile a specific example, you go in the directory and type `make`. In the example that creates libraries you have to type `make static` or `make dynamics` if you want the static or the shared library, respectively.

Compilers

The reference operative system for the course is Linux.
We have two reference compilers. The first is the gnu compiler (g++), at least version 4.8. It is normally provided with any Linux distribution. Check the version with `g++ -v`.

The second C++ compiler is clang, of the LLVM suite, downloadable from llvm.org. Use version 3.1 or higher. You may find it in most Linux distributions. With respect to the gnu compiler it gives better error messages (and it is faster).
You may also try the Intel compiler. Highly optimized for intel processors. You may obtain it for free for personal use only.

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

On line C++ references

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

- www.cplusplus.com: an excellent *on-line reference* on C++ with many examples, adjourned to the new standards `C++11` `and` `C++14`.
- www.cppreference.com: another very complete reference site.
- Wikipedia is also a useful source of information.

Use the web to find answers!

Development tools

The use of IDEs (Integrated Development environment) may help the development of a software. You may use the software eclipse (downloadable from www.eclipse.org and provided by several Linux distributions)
Another good IDE is Code::Blocks.
All examples illustrated in the course will contain a *Makefile* to ease compilation (we will make a lecture on Makefile).

Of course, it is not compulsory to use an IDE. A good editor is sufficient! Personally, I use emacs. Other good editors are: nedit and vim. All support sintax highlighting.

A note on the language used in the course

The course is given in English. Forgive my mistakes. Yet, I may suggest a book for those of you who wish to write 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.

A note on the C++ language

C++ has evolved recently. A new standard called C++11 (and now C++14) has been produced. The previous standard is indicated by C++98.

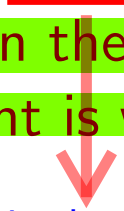
The new standard has introduced major addition and changes to the language (portability of C++98 code is however granted).

All major compilers implement large part of the new standard fully. We will make use of the most interesting features of the new standard in this course. We will use features already supported by gnu compiler g++ (since version 4.8) and clang++ (since version 3.4).

Code documentation

Documenting a code is important. We will 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.



Lesimonare, essere risparmioso

A possible definition of scientific computing

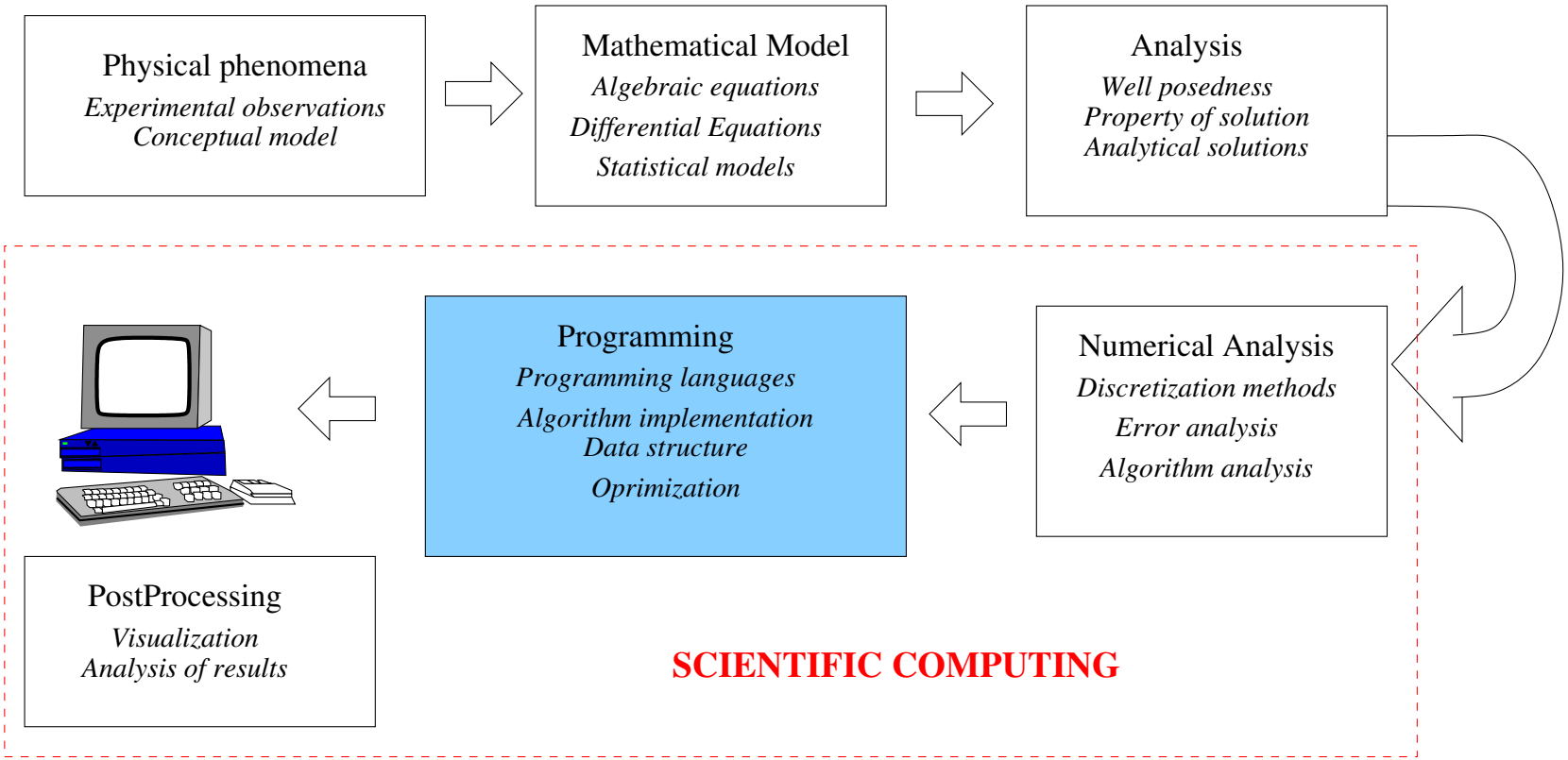
Scientific computing is the discipline that allows to compute in an effective way a mathematical model described by a numerical algorithm.

The main objectives are

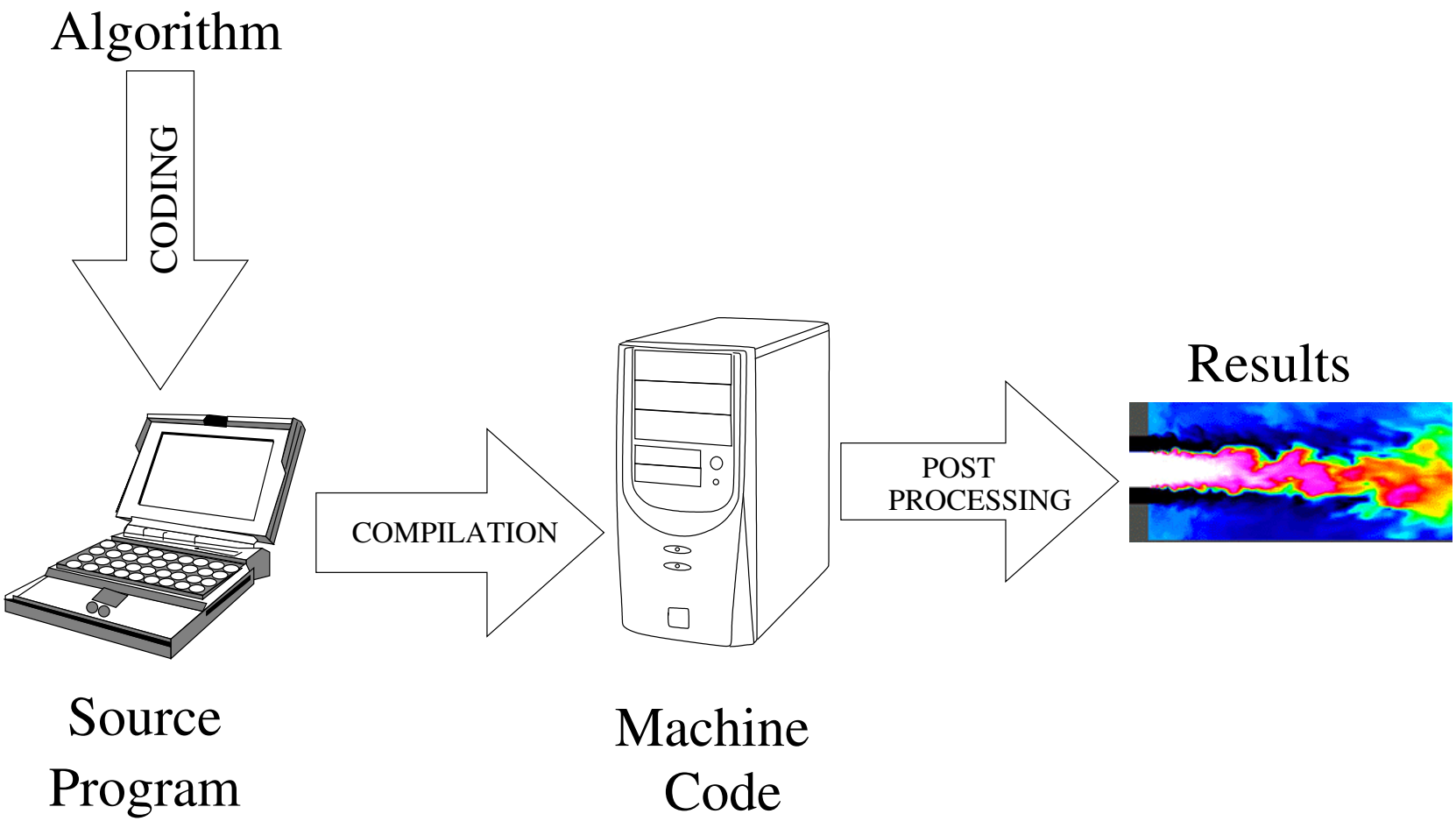
- To find the good compromise between efficiency (in terms of cpu time and memory usage), generality and re-usability of code;
- To control, as far as possible, errors introduced by computation.

To reach this objective we need good knowledge of numerical methods, computing languages and computer architectures.

From physics to computer



From algorithm to result



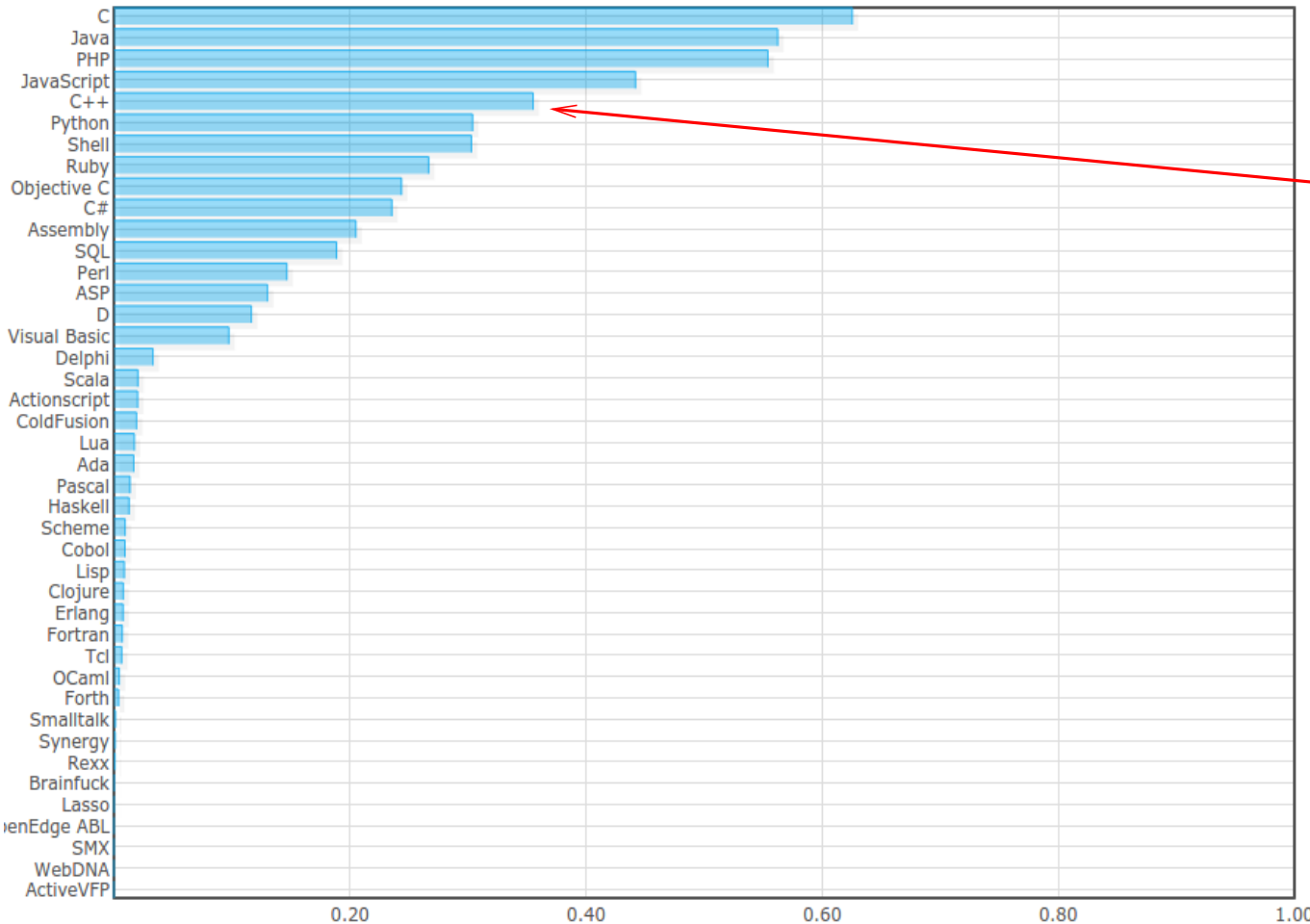
Main programming language types

- Interpreted Instructions are processed sequentially and translated into actions. Examples are BASIC, MATLAB, Python.Simplicity of programming and debugging. Slow code.
- Compiled The entire source code is translated to machine code, creating an executable. Examples are C, C++, FORTRAN. Faster code. Debugging more difficult. The executable is architecture dependent.
- Semi-compiled The source code is translated into an intermediate, architecture independent code (byte-code). The latter is interpreted "run-time" by a "run-time environment". Example: Java.Intermediate efficiency.

C++ usage

Programming language popularity

langpop.com 2015



C
Java
PHP
Javascript
C++
Python
Shell
Ruby
C#
Perl

Fortran

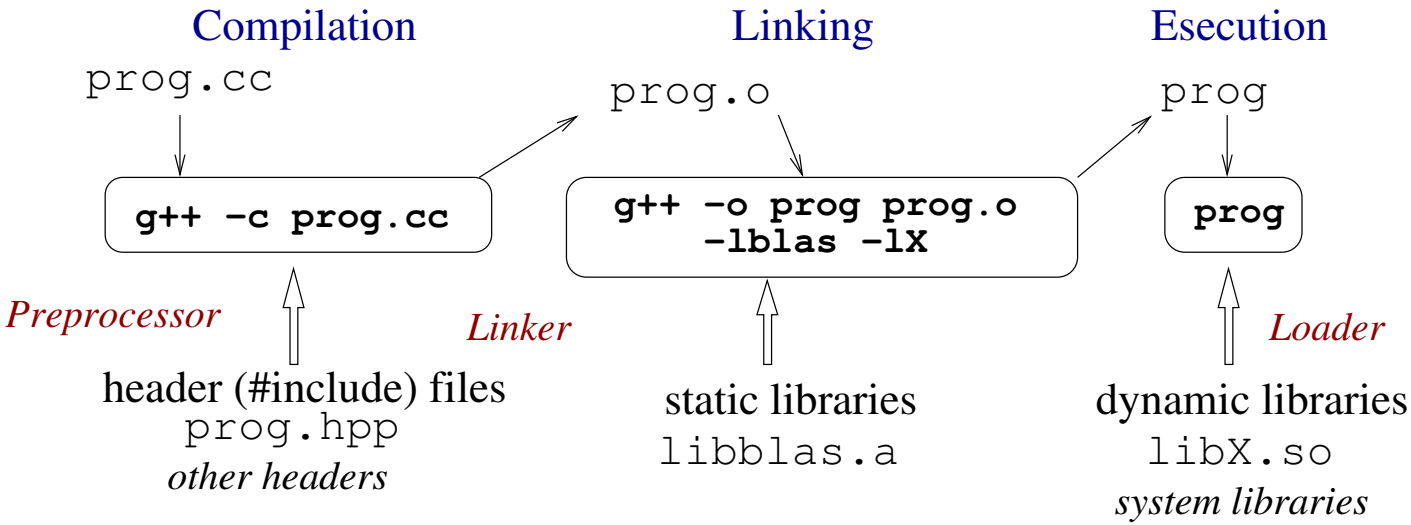
Alternatives for scientific computing

- FortranXX. Mainly procedural programming. Fortran90 has introduced *modules* and *dynamic memory allocation*. There is some support for OO programming. Very good the intrinsic mathematical functions. Produces very efficient coding for mathematical operations.

Object Oriented

- Java. There are quite a few numerical applications, yet Java codes are often not efficient enough for scientific computing. Only OO programming. Good for web computing.
- C. Much simpler than C++, it lacks the abstraction of the latter. Many commercial codes for engineering simulations are written in C. The Linux kernel and the GNU OS API is in C.
- Python Very effective in building up user interfaces and connect to code written in other languages. Many modules for numerics, like *PyNum*. Its use in scientific computing is on the rise (see the FeNics project).
- Matlab There is support for OO programming and can be compiled (if you buy the compiler), but it is essentially an interpreted language. It's free "clones" *Octave* and *Scilab* are good alternatives.

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



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

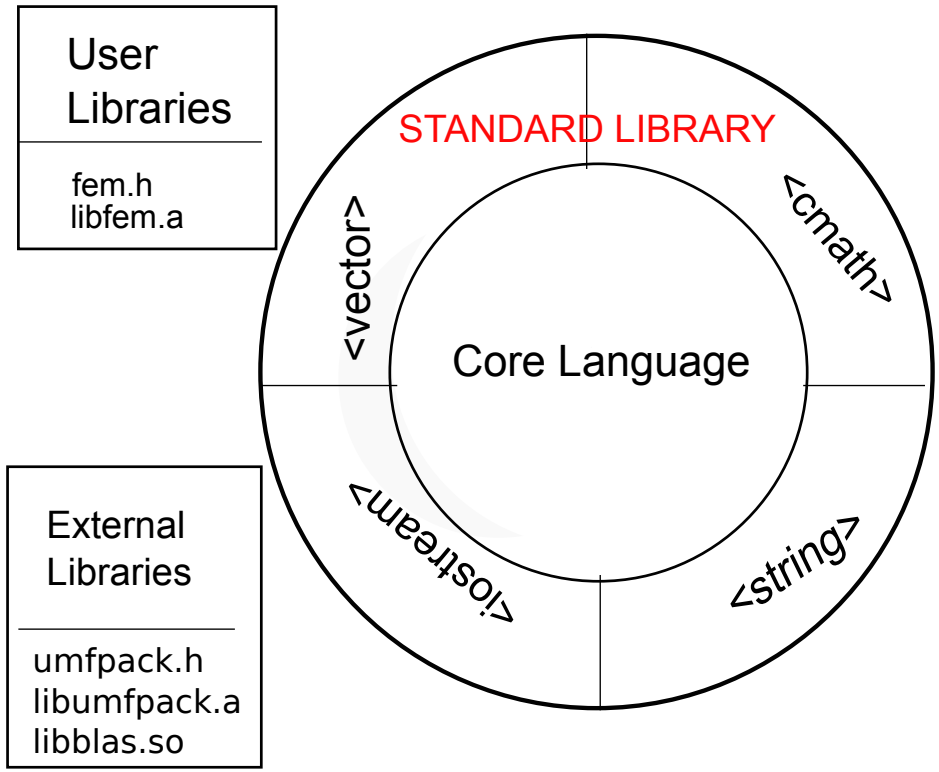
MAIN g++ OPTIONS (some are in fact preprocessor or linker options)

-g	For debugging	-O[0-3]	Optimization level
-Wall	Activates all warnings	-Idirname	Directory of header files
-DMACRO	Activate MACRO	-Ldirname	Directory of libraries
		-lname	link library
-std=c++11x	activate c++11 features	-std=c++14	activate c++14 features

Compilation unit

A C++ program is normally formed by several source files (*.cpp) and related user defined or system header files (*.hpp).
The header files are normally kept either in the same directory of the source files or, more frequently, in a separate directory (typically called `include`).
Each source file contains the code that implements specific functionalities and, together with the set of header files it includes, is called a compilation unit. E' l'unione di source file e header files inclusi
The compilation process treats each compilation unit independently, producing object files (*.o).
The linker will gather the information of the object files to produce an executable program. The executable, when run, will invoke the loader, which loads the required shared libraries.

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 functionality is provided by the *Standard Library* and require to `include` the appropriate `header` file (also called `include` file). For instance, if we want to perform i/o we need to include `iostream` using **#include** <iostream>.

List of C++ standard header files (C++98)

<*algorithm*>	general algorithms
<*bitset*>	set of Booleans
<*cassert*>	diagnostics, defines *assert*() macro
<*cctype*>	C-style character handling
<*cerrno*>	C-style error handling
<*cfloat*>	C-style floating point limits
<*climits*>	C-style integer limits
<*clocale*>	C-style localization
<*cmath*>	mathematical functions (real numbers)
<*complex*>	complex numbers and functions
<*csetjmp*>	nonlocal jumps
<*csignal*>	C-style signal handling
<*cstdarg*>	variable arguments
<*cstddef*>	common definitions
<*cstdio*>	C-style standard input and output
<*cstdlib*>	general utilities
<*cstring*>	C-style string handling
<*ctime*>	C-style date and time
<*cwchar*>	C-style wide-character string functions
<*cwtype*>	wide-character classification
<*deque*>	double-ended queue
<*exception*>	exception handling
<*fstream*>	streams to and from files
<*functional*>	function objects
<*iomanip*>	input and output stream manipulators
<*ios*>	standard *iostream* bases
<*iosfwd*>	forward declarations of I/O facilities
<*iostream*>	standard *iostream* objects and operations
<*istream*>	input stream
<*iterator*>	iterators and iterator support
<*limits*>	numeric limits, different from <*climits*>
<*list*>	doubly linked list
<*locale*>	represent cultural differences
<*map*>	associative array
<*memory*>	allocators for containers
<*new*>	dynamic memory management
<*numeric*>	numeric algorithms
<*ostream*>	output stream
<*queue*>	queue
<*set*>	set
<*sstream*>	streams to and from strings

C++11/14 header files

The new standards have added many more header files. The full list is to long to write here.
You may find it in cppreference.com with full explanations.

C and C++

Also the C language provides many `header files`. Most of them have been *inherited* by C++, but the name has been changed using the following rule

C Header file	C++ header file
name.h	cname

For instance the C header file `assert.h` is available in C++ under the name `cassert`. Standard header file are normally included using **#include** <file> and not **#include** "file".
The latter format is usually reserved to user defined header files (the way the preprocessors searches directories to find the header file is different in the two cases).
In C++ the suffix `hpp` is preferred to `h` (but it is not compulsory).
Note however the the standard C++ header files have no extension!.

The std namespace

Names of standard library objects are in the std namespace. Therefore, to use them you need either to use the full qualified name, for example std::vector<**double**> or bring the names to the current namespace with the **using** directive:

```
#include <cmath> // introduces std::sqrt
...
double a=std::sqrt(5.0); // full qualified name
...
using std::sqrt; //sqrt in the current namespace
...
double c=sqrt(2*a);// OK
```

With **using namespace** std you bring all names in the std namespace into the current scope. Beware: use it with care.

Some nice utilites

To be able to input parameters and data from files, or from the command line, we will use the utility GetPot.

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 to gnuplot, that allow it to be called from within a program is gnuplot-iostream.

You may also try xmgrace for simple 2D plots.

The `main()` Program

The main program defines the entry point 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 execution.
Their processing is however a little cumbersome. In our course we will make use of the GetPot utility to make life easier.

Scomodo, goffo, impacciato

An example of command parsing

```
GetPot    cl(argc, argv);
if( cl.search(2, "-h", "--help") ) printHelp();
bool verbose=cl.search(1,"-v");
// Get file with parameter values
string filename = cl.follow("parameters.pot","-p");
```

What does `main()` return?

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

The integer returned by the `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".
Another way to set the return status is by using the `exit()` statement (you must include the `cstdlib` header in this case).

Two simple examples

A simple C++ program. In `Examples/src/SimpleProgram/main_ch1.cpp` A program that computes $\sum_{i=n}^{m} i$.
HeatExchange. A simple 1D finite element program.

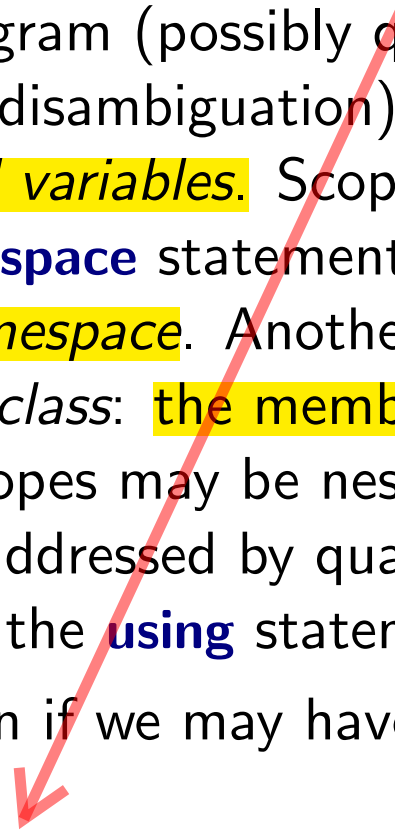
Some nomenclature

Signature di una funzione = nome della funzione + parametri in ingresso
Nella signature non è incluso il tipo restituito
--> non posso eseguire un overload modificando solo il return type

- Identifier An identifier is an *alphanumeric string* that identifies a variable or a function uniquely. The identifier of a variable is its *name*, for a function is its *signature*. A *qualified identifier* adopts a *full qualified name*.
- Symbol The internal translation of an identifier.
- Name A generic term indicating an alfanumeric string that identifies entities like variables or sets of overloaded functions. Names in C++ cannot begin with a numeric character and are case sensitive. It is important to remember that hiding applies to names. A name is fully qualified if it contains the name of its enclosing class or namespace using the scope resolution operator, i.e. `std::cout` is qualified, while `cout` is not. A name cannot be equal to a `keyword` of the language (e.g. I cannot have a variable called `while`).

Presenti nel file oggetto (--demangle)

- Scope In C++ a scope is a part of a program consisting of statements enclosed by a pair of braces ({}), with the exception of the *global scope* that is outside all braces. Variables defined in the global scope are *global variables* and are *visible* everywhere in the program (possibly qualified by the global scope identifier :: for disambiguation). Variable defined in a local scope are *local variables*. Scopes may be given a name, through the **namespace** statement, and a named scope is also called a *namespace*. Another type of named scope is introduced by a *class*: the members of a class are in the scope of the class. Scopes may be nested and names in named scopes can be addressed by qualifying them or brought to the local scope by the **using** statement.
- Namespace A named *scope* (even if we may have unnamed namespaces!).



Tutti i file vengono compilati separatamente,
le variabili di un file non vengono viste da un altro
a meno che non siano globali
(static vs extern, vedi appunti)

- Object. With the term *object* we refer to a memory location (which typically stores a variable). The process of creating an object is called *construction* or *instantiation*, 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 is a *member* of a class if it belongs to the definition of that class. A variable is *constant* is its content cannot be changed.
- Hiding Mechanism by which a name in an enclosed scope hides a name (function) declared in the enclosing scope. The object associated to the hidden name may still be accessed using its fully qualified name.

- Operators An operator is a particular function that combines one or two arguments (in one case three arguments) and returns a value. For instance + can be a unary operator, like in c=+5, or a binary operator (addition), like in c=a+b. C++ allow overloading of most operators. An operator has a preferred signature and return type that should be complied with when overloading. For instance, the signature and return type of the binary addition is
  T & **operator** + (T **const** &, T **const** &). A full list of C++ operators is easily found on Wikipedia!.
- Expressions Combination of *operators*, function calls, and *variables* (or constants) that produce a value, i.e. 5+a where a is a int, is an *integer expression*.

- Constant expression. A constant expression is an expression that can be computed *at compile time*. In C++11 constant expressions may be declared with the `constexpr` statement.
- Literal. Literals are the simplest case of constant expressions. They are used to express particular values within the source code of a program: in **double** a=2.3;, 2.3 il a literal. So it is better to write **const double** a=2.3;, or, even better (C++11) constexpr **double** a=2.3;.

Declaration and definition

Dichiarare una funzione significa specificare signature e return type
Definire una funzione significa fornire non solo signature e return type ma anche il corpo (codice, cioè cosa fa)

- Declaration. It introduces one or more identifiers into a program and specifies their *type*. It allows the compiler to determine the *size* of an object of the given type. Declaration may occur more than once in a program, but they must be identical.
- Definition. It means provides the information necessary to create an object in its entirety. Defining a function means providing a function body. Defining a variable means specifying its value etc. A definition is also a declaration.
- Istance. The moment when an object is created (the meaning is different for templates).
- Initialization. It provides the initial value to an object during its construction.
- Assignment. To provide a value to an already constructed object.

- function or templates parameters. The type of the variables (for functions) or types/values (for templates) given in the declaration:

```
template<class T, int N> class myclass;
void fun(double& a, int b);
```

Here, T, N, a and b are parameters.

- function or templates arguments. The name/value of the variables (for functions) or types (for templates) given in the instance:

```
myclass<double, 3> c;
int I=9;
fun(7.0, I);
```

Here **double**, 3,7.0 and I are arguments.

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 (default = cout)

Readdressing standard input output and error

In the linux `bash` shell i/o can be redirected. Suppose `main` reads form the standard input and writes to the standard output.

```
main> a.txt # writes stdout on file a.txt stderr on terminal
main <b.txt # reads stdin from file b.txt
main  2>error.txt # stderror in error.txt
main >a.txt 2>&1 # both stdout and stderr in a.txt
```

A little introduction to floating point numbers

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

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

where

β	basis (typically 2)	t	precision E' il numero di cifre decimali
m	mantissa, $0 \leq m \leq \beta^t - 1$	e	exponent, $e_{min} \leq e \leq e_{max}$

Analogously,

$$y = 0.d_1 d_2 ... d_t \times \beta^e \quad d_1 \neq 0, \quad 0 \leq d_i \leq \beta - 1$$

All floating point numbers which can be represented 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.

Rounding off

A real number $x \in \text{range}(F^*)$ is represented in a computer by $\hat{x} = fl(x) \in F^*$ and $|x - \hat{x}|/x$ is the (relative) rounding error. If $x \in \text{range}(F^*)$ then $\hat{x} = fl(x) \in F$. In general,

$$\frac{x - \hat{x}}{x} = \Delta, \quad \text{with} |\Delta| \leq u = \frac{1}{2}\beta^{1-t}$$

u is the roundoff unit and is linked to the machine epsilon ϵ_M by

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

ϵ_M is the smallest positive floating point number by which $fl(1 + \epsilon_M) \neq 1$.

Note: $0 \in F$ and $1 \in F$!.

IEEE arithmetic

modificato

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

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

$F_{min} \in F^*$ 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 indicates the result of an invalid operation $(0/0, \log(0)$ etc) and we have $x \,\text{op}\, NaN = NaN$, $Inf - Inf = NaN$ e $0/Inf = NaN$.

Floating point exceptions

The standard prescribes that in normal situations an invalid operation or an over/underflow do not stop computations, but it produces the special numbers illustrated before.
However, the processor may record if an invalid operation (normally called floating point exception has occurred, so that the user may trap it. intrappolare, bloccare
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

$$\boxed{\frac{|y - \hat{y}|}{|y|} \leq \delta}$$

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

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

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

$$\hat{y} = f(x(1 + \Delta)) \leq C(x)|\Delta|$$

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

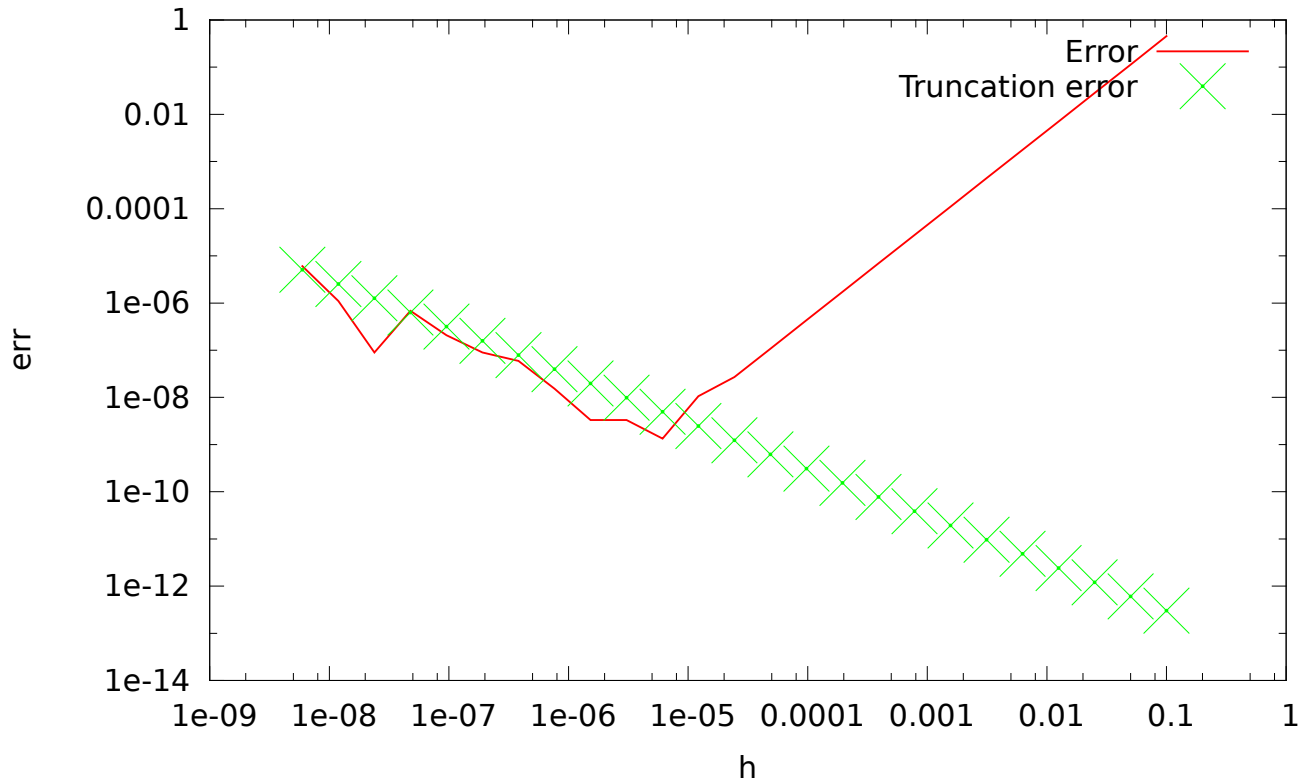
$$\hat{\delta}_f(x) = \frac{f[(x+h)(1+\Delta_1)] - f[(x-h)(1+\Delta_2)]}{2h}$$

If $f'(x) \neq 0$ we may roughly estimate

$$|\hat{\delta}_f(x) - \delta_f(x)| \simeq |xf'(x)u|$$

u being the roundoff unit.

In the example Examples/src/FinDiff we compute the numerical derivative of $100e^x$ at the point $x = 1$ and plot the error $|\hat{\delta}_f(x) - f'(x)|$ for different values of h, together with the estimated forward truncation error.



Inizialmente, l'errore di troncamento è irrilevante rispetto all'errore totale: sto usando un h troppo grande
Per valori di h piccoli (< 1e-05): l'errore commesso è inferiore alla stima dell'errore di troncamento, dunque la differenza è tutta dovuta all'aritmetica floating point, h non c'entra più nulla --> 1e-05 è il valore ottimale

Another example: the zeros of a quadratic

The zeros of $ax^2 + bx + c = 0$ may be computed by the standard formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

What happens if $b >> ac$... We have $\sqrt{b^2 - 4ac} \simeq |b|$. One of the operations implies cancellation.
Have a look to Examples/src/QuadraticRoot

Numeric limits

C++ allows you to interrogate the characteristics of the numeric types as implemented in your machine. Non only floating points, but also integral types. In particular, we may look for the machine epsilon. We need to use the module of the Standard Library loadable by including `<limits>`.

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

See also a complete example in Examples/src/numeric_limits

Initialization: old and new (C++11)

C++11 has introduced a new way of initializing variables extending the parameter list initialization that was available before only for fixed dimension arrays

```
int i(0); //i is initialized to 0
int j=0; // as before: equivalent to j(0)
int k{0}; // only c++11
int f={0}; // only c++11
// initializes an array of 3 elements
double c[]={0.,1, 3.} // OK
struct data{ int a; float b;}// Declaration
data d={3,4.0};// only c++11
// a vector<double> of 3 values (only c++11)
std::vector<double> v{4.,5.,6};
// a vector<int> of 2 values (only c++11)
std::vector<int> w={1,2};
```

An important note

Never assume that a variable is initialized automatically. Always initialize variable explicitly, it 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.
double * x=nullptr;// Since c++11 (much better!)
int * ip=new int[20]; //Ok pointer to an array of int
double * pp;// NO!
```

Beware of floating point comparisons!

Floating points have discrete values. So camparing them is always very critical. The statement

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

is dangerous. Maybe the test will be 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 (see `numeric_limits`).

Implicit and explicit conversion

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

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

Note: It is safer to use explicit conversion and it is more efficient to use static casting.

C-style case, e.g. z= (**double**) a; , is allowed but discouraged in C++.

Special conversions

```
int a=5; char A='A'
bool c=a; c is true
a=A; a is the ASCII code of 'A'
```

Any non null value of POD is converted to **true** The null pointer is converted to `false`, a non-null pointer to `true`.

Note: This rule may be useful.

Scope

We have already give the definition of scope. We recall some important facts about them:

- Scopes are 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.
- 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 instantiated, unless it has been declared *static* in a function.

An example about scope

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

typedef

The command **typedef** creates an alias to a type and it is very useful to save typing or having to remember complex types.

```
typedef unsigned int Uint;
typedef vector<float> Vf;
typedef double Real;
typedef Real myvect[20];
...
Vf v;// v is a vector of floats
myvect z // z is an array of 20 doubles
```

A simple rule If you take out `typedef` you obtain the declaration of a variable. Typedefs are useful if you have long type names: a vector of a vector of pointers to double:

```
typedef vector<vector<double*> > Vvdb
```

the using keyword (C++11)

C++11 has introduced an new keyword, that somehow replaces **typedef** and is more general and intuitive!

```
using Real=double; // equivalent to typedef double Real
using func = void (*) (int,int); // func is a pointer to function
template<typename T, int N> Point; // a template for points
template<typename T>
using Point2D = Point<T,2>;
Point2D<double> myPoint; // Defining a Point<double,2>
template<class T> using ptr = T*;  // ptr<T>'is  an alias for poi
ptr<int> xPtr; // definint a pointer to int.
```

Prefer **using** to **typedef**!.

the auto keyword (C++11)

The new C++11 standard has introduced a new keyword to simplify the handling of complicated types and also ease generic programming. 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 converts to a the type, omitting qualifiers: you should add & or const if needed.

The rules for auto

The type deduction for **auto** is the same as for template parameters (apart some exceptions). We will not dwelve into the matter now. I provide only examples, recalling the main rule: lvalue reference specification is stripped!. We will go back into the matter later on in the course. Anyway, in the 98 per cent of cases auto does what you want.

```
double & fun(double); // a function returning a reference.
auto a =fun(6.0)// a is a double not double&
auto & b =fun(6.0)// b is a reference!
std::vector<int> in;
for (auto i : in) i=3; // fills the vector with 3
// i is here an int
```

The actual rules for auto type deduction are rather complex, we will mention them when necessary!

Extracting the type of an expression (C++11)

You are probably aware of the command **sizeof**(), which returns the size of an expression or of a type (in bytes). For instance **sizeof**(**double**) returns 8 (in most systems). With C++11 we can finally 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
```

This new feature is handy for generic programming.