Advanced Methods for Scientific Computing (AMSC)
Lecture title: A (not so) brief overview of C++

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2025/2026

Why C++

C++ is

- Reasonably efficient in terms of CPU time and memory handling;
- In high demand in industry: it is now the 2nd most popular language according to TIOBE Index;
- 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*.

Alternatives for scientific computing

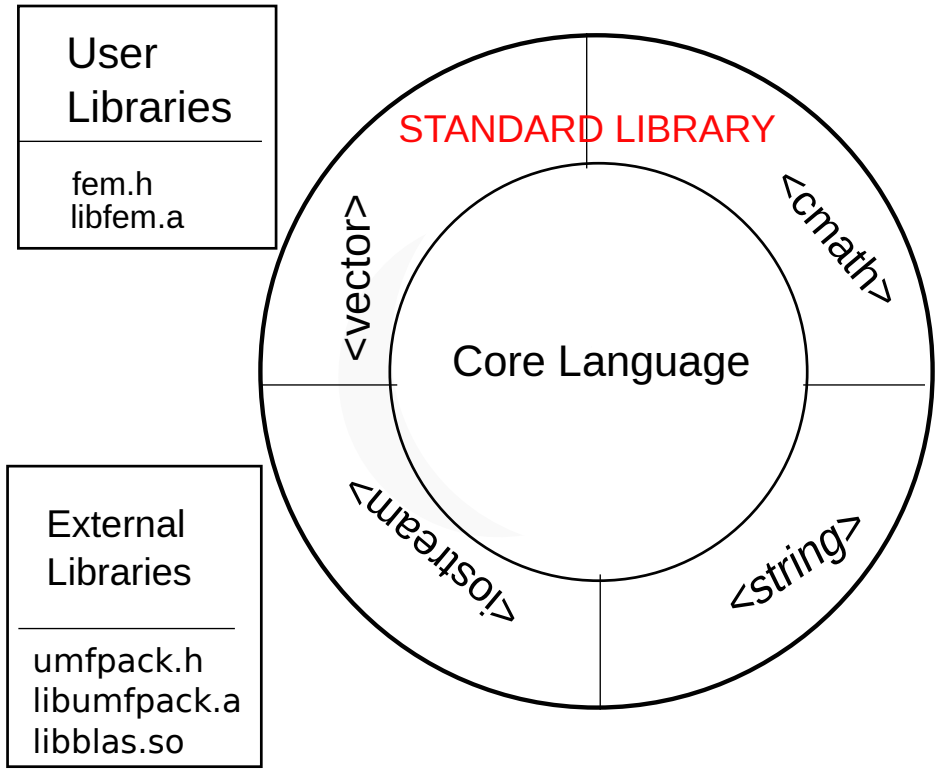
- Python Very effective in building up user interfaces and connect to code written in other languages. Many modules for scientific computing and statistics, as well as machine learning. Its use in scientific computing has been on the rise.
- FortranXX. Mainly procedural programming. Fortran90 has introduced *modules* and *dynamic memory allocation* and some support for OO programming. Very good the intrinsic mathematical functions. Produces very efficient coding for mathematical operations.
- C. Much simpler than C++, it lacks the abstraction of the latter. The Linux kernel and the GNU OS API is in C.
- Matlab/R They are essentially interpreted languages. Octave is a good free alternative for Matlab, with nice interface to C++.
- Julia A new interpreted language that exploits LLVM infrastructure to perform just-in-time compilation to gain efficiency comparable to a compiled language. On the rise in the scientific computing world.

Important libraries for scientific computing

Unfortunately C++ has not the nice development environment of python (the new module system introduced in C++20 will change the situations in the next years). You have anyway available many libraries to support scientific computing¡ for instance:

- Boost libraries. A suite of libraries that work well with the C++ Standard Library. For instance the Boost Graph library.
- Eigen A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Very fast and well supported.
- PETSc.The Portable, Extensible Toolkit for Scientific Computation for the scalable (parallel) solution of scientific applications modeled by partial differential equations (PDEs).
- Trilinos. A collection of reusable scientific software libraries: linear solvers, non-linear solvers, transient solvers, optimization solvers, and uncertainty quantification (UQ) solvers.
- Deal.II. A C++ software library supporting the creation of finite element codes and an open community of users and developers.

The structure of the C++ language



C++ is a highly modular language. The core language is very slim, being composed by the fundamental constructs 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>.

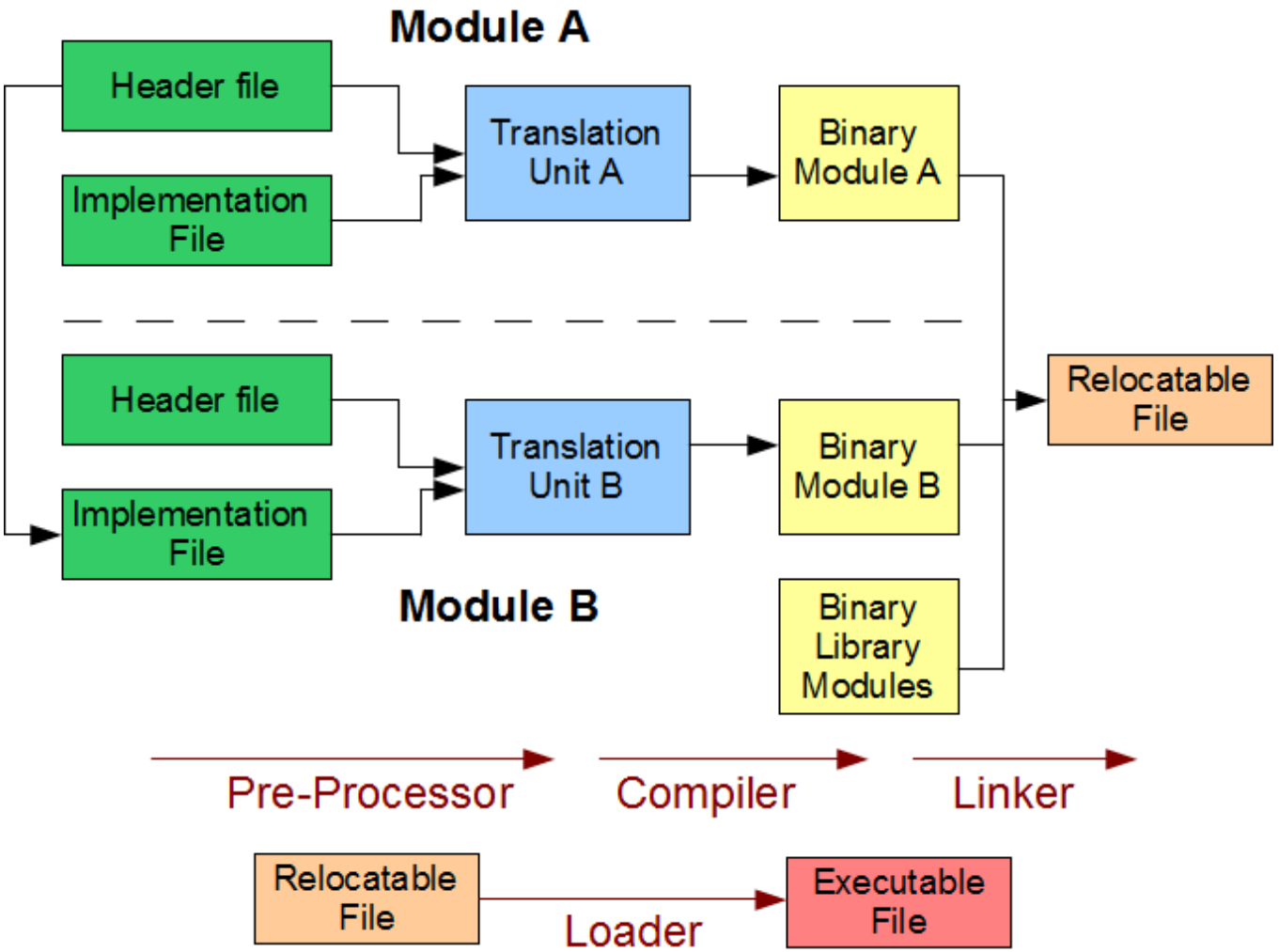
Software organization

Being a compiled language derived from C, C++ shares in good part the structure of a C code:

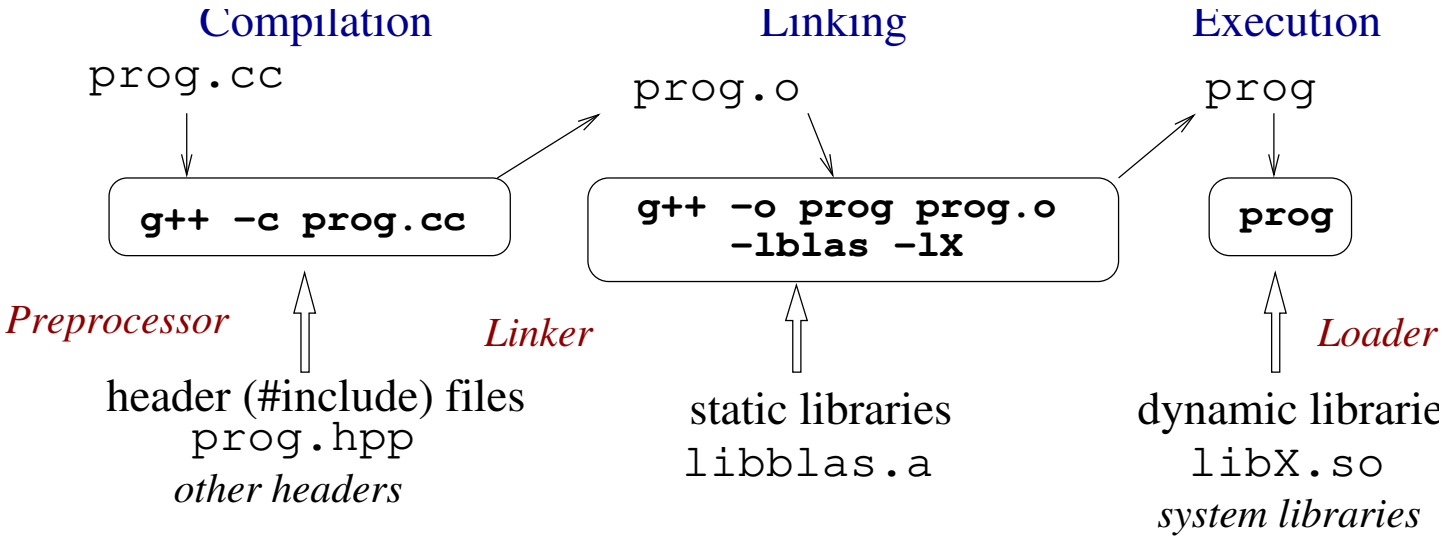
- header file (typical extension `.hpp`). Contain material needed to the compiler to deduce types and instantiate templates and inline function: forward declarations, definition of templates and inline functions, forward declaration of global variables... They are included by all translation units than need to use the declared items.
- source files (typical extension `cpp`), also called implementation files. They contain the definitions of functions and class methods (when not inlined). One of them contains the `main()` program. They `include` the needed header files.

A translation unit is formed by a source file and all the included header files and is the basic unit of compilation.

The compilation process



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.

Modern C++

The term modern C++ indicates the language since standard 11 ("C++11 looks like a new language". B. Stroustroup). It has:

- Safer memory handling: the introduction of smart pointer allows implementing the RAII (resource acquisition is initialization) principle easily also for polymorphic objects.
- Uniform initialization: everything can be initialized with braces ({}).
- Extended automatic deduction of types.
- Improved functional programming: lambda expressions.
- Improved support for generic programming.
- Direct support of parallel programs with parallel algorithm.
- Concepts and ranges (introduced in c++20)

C++ version for this course

We will refer to C++20, even if we will not use many of the nice C++20 features. To activate C++20 you need to use the `-std=c++20` option of the compiler, for instance:

```
gcc -std=c++20 -O3 -o main main.cpp
```

which activetes also aggressive optimization with the `-O3`.
One of the characteristics of C++ is backward compatibility. So, you can mix "modern" and "old" C++ (a part a minor things made obsolete).

Some subtle differences with C

C++ is highly compatible with C and it inherits most of its syntax. A part the major additions (classes, templates, etc.) there are some subtle differences

- We have namespaces to reduce the possibility of name clashes: std::vector<**double**> is in the namespace std;
- A function taking no argument is declared as f() and not f(**void**);
- Function overloading: the argument's type (and some qualifiers...) is part of the function identifier: **double** f(**int**) is a different function than **double** f(**double**), even if they share the name;
- The syntax of a **struct** is slightly different than in C: **struct** in C++ is basically a synonym of **class**;
- Operators may be overloaded (indeed it is a particular case of function overloading). You can extend most operators to your classes.

Some subtle differences with C

- Constness: variables may be declared **const**, to specify that are not meant to be changed (this makes the program safer and may induce optimizations);
- Variable may be declared **constexpr** to indicate that they are in fact immutable constant expressions:
  constexpr double pi=3.1415;
- We have references (& and &&) to indicate aliases to existing objects. With **double** & a=c; we make a an alias of c. References are used (in place of pointers) to "pass by reference" a function argument. The initialization of a reference is called binding and cannot be broken (you cannot have an unbound reference nor change the bindings).

Namespace

Namespaces are a C++ feature to create a more modular architecture and reduce the possibility of name clashes (having two different objects with the same name in the same scope.). They indeed create a named scope

"foo.hpp"

```
namespace apsc
{
  inline double triple(double x){return 3*x;};
  inline double z = triple(9);  // a namespace variable!
}
```

"main.cpp"

```
#include "foo.hpp"
int main()
{
  double x = triple (5.); // ERROR!
  double y = apsc::triple(5.);// OK! using fully qualified name
  using namespace apsc; // inject names in the current scope
  double z = triple(5.);// OK!
}
```

Nested Namespaces

Namespaces can be nested

```
namespace apsc
{
  namespace utilities
  {
    double fun(double * x);
  }
}
...
auto a= apcs::utility::fun(10.45);
```

The std namespace

Names of standard library objects are all 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 scope 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 scope
...
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 and not in header files!

A curiosity for the ones who knows C

C++ is highly compatible with C and has inherited a lot of header file of the C standard library.

The rules to recognise a C header file ported to C++ is

C Name: header.h $\rightarrow$ C++ name: cheader

C++ standard library header file have no extension.

Do not use a C standard header file when there is an equivalent C++ standard header file.

The `main()` Program

The main program defines the only 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 launching.

Their parsing is however a little cumbersome. The use of GetPot utility makes parsing easier, as we will see later on.

Why does `main()` return an integer?

The return integer may be set using the return statement. If the **return** statement is missing, by default the main program returns 0.

The integer returned by `main()` is called *return status* and may be interrogated by the operative system. For instance, you may decide to take a particular action depending on the return status value. By convention, 0 means "executed correctly".

To terminate a program not from the main you can use std::exit() (you have to include cstdlib), which accepts a return status as argument.

Basic types

Most inherited from C

- **float**. 32 bits floating point (es. 3.0f);
- **double**. 64 bits floating point(es. 3.0);
- **long double**. $>$ 64 bits floating point (es. 3.0L);
- **int** A signed integer ($\geq$ 32 bits) (es. -3);
- **unsigned int** An unsigned integer (es. 3u);
- **long unsigned** Long and unsigned: (es. 3ul);
- std::size_t The integer type used to address vectors and arrays;
- **char** A character, guaranteed to be 1 byte (8 bits);
- std::**wchar_t**. Long character (for multilingual);
- **byte**. A byte (8 bits).

Fixed width integers

If you want to know your bits, you may use the fixed width integers (<cstdint>)

- std::int8_t. 8-bit signed integer;
- std::uint8_t. 8-bit unsigned integer;
- std::fast_16_t. fastest signed integer type with width of 16 bits;
- std::intptr_t. signed integer type capable of holding a pointer to void;
- std::int_least16_t. smallest signed integer type with width of at least 16 bits.

Full list in this page

Other important C++ standard types

- std::**string**. The C++ string type (<**string**>);
- std::array<T,N>. Array of N elements of type T (<array>);
- std::vector<T>. Dynamic vector of elements of type T (<vector>);
- std::complex<T> Complex number on the numeric type T (<complex>);
- std::tuple<T...> Fixed size collection of objects of different type (<tuple>);
- std::bitset<N>. A fixed-size sequence of N bits (<bitset>) ;
- i/o streams. To do basic i/o (header <iostream>);
- And many others (set, map,....).

const qualifier, constexpr, (l-value) references

```
double x; // a double, not initialises, vulue undetermined
double y{10.0};// a double initialised to 10 (direct init.)
double z=-9.;//a double initialised to -9 (copy initialization)
int const a=10; // a cannot be changed (in principle)
a=20; // ERROR! a cannot be changed
constexpr pi=3.14;//pi is a constant expression. It's immutable
double & w=z; // w is an ALIAS of z;
w=-9.; // now also z is equal to -9.
double const & l=z;// another reference to z, it's value can't be changed
l=-7.8;//Error! l is const.
const_cast<double &>(l)=-7.8;// Now I can change value (don't do it)
```

const: a variable that cannot be changed (unless forced);

constexpr: an immutable value;

reference: an alias to an object (possibly const).

The use of (l-value) references

L-value references, like **double**& are very important in C++ programming and essential in C++ for scientific computing. A main usage is as function parameters (and sometimes return type). Passing a reference instead of a value help saving memory, since you refer to an existing object and dont make a copy. 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 present the binding rule of l-value references in the next slide.

A first overview of l-value reference bindings

```
double f(double & x); //function changes the value of x
double f(double const & x);//an overload. x is not changed by the function
double g(double const & x);// g has only this overload
double h(double & x); //function changes the value of x
...
double y=10; // copy initialization
double const z{100.};// direct initialization
constexpr double pi=3.14;
f(y); // calls f(double&). The value of y has changed
f(z); // calls f(const double &). z is unchanged
f(pi); // calls f(const double &)
g(y); // ok
g(z);// ok
g(pi);// ok
h(y); // The value of y has changed
h(z);// ERROR z is const
h(pi);/// ERROR pi is constant
```

This is just an example to explain bindings. Non const l-value reference can bind only to "values that can be changed" (called l-values). A const l-value reference can bind to anything, but the non-const version is used if viable.

What about the && stuff? (rvalue references)

In the Examples you will sometimes find the use of rvalue references:

```
double fun (std::vector<double> && x);
```

They are references which with a particular binding rule: they bind only to temporary values (rvalues) and are crucial in implementing efficient use of dynamic memory when dealing with large objects:

```
std::vector<double> createVector(std::size_t n);
double fun (std::vector<double> && x);//I move the vector
double fun (const std::vector<double> & x);
....
std::vector<double> x(100,1.0);
fun(x)// fun (const std::vector<double> & x) called
fun(createVector(100));//fun (std::vector<double> && x);
```

In this course, we will skip this advanced topic (if you are interested, ask in the forum and I will provide some material). A full example of reference bindings is in the Bindings folder.

pointers, smart pointers and addresses

Pointers are variables capable to store the address of an object of a given type and return its value via the dereferencing operator $*$.

In modern C++ "bare pointers" are seldomly used oan only for "watching a resource". If a pointer is used to handle a resource on the heap store (dynamically) you should use an smart pointer. Smart pointers own the resource they point to, which is destroyed automatically when the pointer goes out of scope. We will have a specific lecture.

```
double a=10.5;
double * b=&a;// b points to the address of a
*b+=0.1;// a now contains 10.6;
double const * z=&a;// a pointer to const
*z=9.0;// Error z is a pointer to const!
```

We will use bare pointers mainly to interface with C-type utilities.

The **void** *

When interfacing with C code the "pointer to void" (or "void pointer") becomes handy. **void** * is a particular pointer type that represents just a memory address with no datatype attached (or, better, the type is **void**).

An important characteristics is that any pointer is implicitly convertible to the void pointer.

```
void fun(void * p, int size);// takes a buffer of size bytes
...
double a;
std::vector<double> v;
std::array<int,5> b;
fun(&a,sizeof(double));
// note the difference!
fun(v.data(),v.size()*sizeof(double));
fun(b.data(),b.size()*sizeof(int));
```

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)
```

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

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.

We will see how it is possible to define implicit conversion for user-defined types

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 (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's 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 auto keyword

In the case where the compiler may determine the type of a variable automatically (typically the return value of a function or the 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 provides the *basic type*, omitting qualifiers and references, i.e. you must add & or/and `const` yourself if you need them: **auto** &, **auto const** &.

Deducing variable type

You should already know the **sizeof** operator, which return the size in the stack memory of a variable or an expression, in bytes: **sizeof**(**double**) is 8 in a 64 bit machine. Modern c++ has also a tool to return the type of a variable:

```
double a;
decltype(a) x; // x is a double!
```

It may look useless, but in fact is rather handy in generic programming (templates), when the type of a variable may not be known at programming type (it will be at compile time!).

I/O streams

C++ provides a sophisticated mechanism for i/o through the use of streams. The 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 stream objects 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)

cerr and clog

`cerr` and `clog` are by default addressed to the standard error (channel 2 in Unix), `cout` to the standard output (chennel 1 in Unix). But, you can redirect the channels at OS level:

```
./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 #redirects only stdout (equiv. to 1>)
```

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.

other i/o facilities

We still have also the i/o facilities inherited from the C language, like printf. You need to include <cstdio>. However you cannot mix C streams and C++ streams.

The output form of a C++ stream can be modified by setting format flags, or using stream modifier.

```
std::cout.setf(std::ios::right); //using flags

//using manipulators
std::cout<<std::setprecision(10)<<a;
```

A Note: Since c++20, C++ offers sophisticated tools for formatting i/o, see here. In this course we will not address them.

Enumerators

I assume that you already now about enumerators:

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

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};
...

Color color = Color::GREEN;
...
if (color==Color::RED){
//  go here if the color is red
}
else
{ // here if not red
}
```

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

C++ Arrays and Vectors

With arrays normally we indicate a data structure with a linear and contiguous data storage. In C++ we have different arrays of different types

- 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] (try to avoid using them)
- Fixed size arrays of the standard library. std::array<**double**,5> c
- The vector container of the standard library: std::vector<**double**> c, which supports dynamic memory management.

A warm suggestion: use arrays and vectors of the standard library. They are safer and can be interfaced with their C counterpart easily.

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>`.

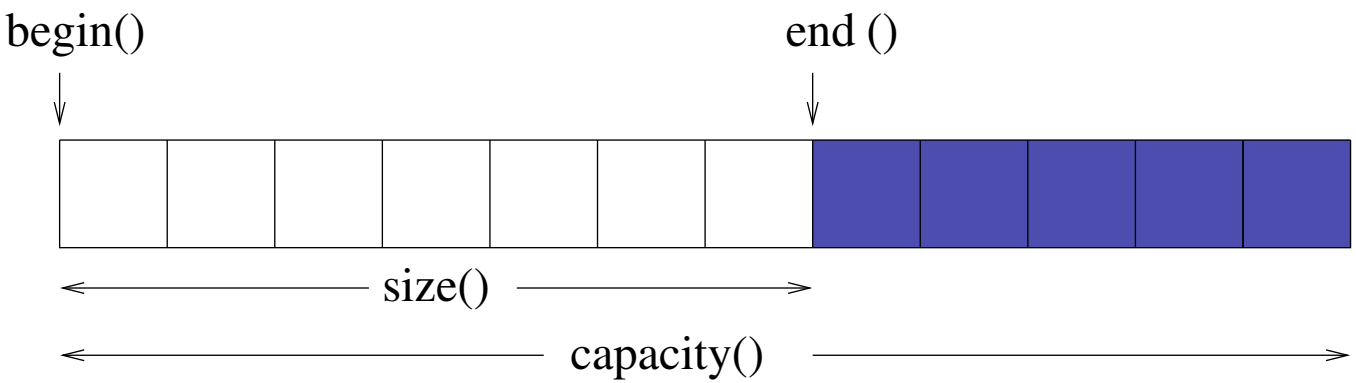
Computational complexity of main operations on vector<>

Random access	O(1)
Adding/deleting element to the end	O(1)[2]
Adding/deleting arbitrary position	O(N)

[2] 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.



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 a. `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
vector<int> b{10,3}; // initializer list
vector<int> c={10,3}; // initializer list
```

Size is 2 and capacity at least 2.

Automatic deduction of template parameters

The compiler may deduce the type of the element from that of the initialization values.

```
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 (and arrays)

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.

A note: If the size is fixed consider using std::array<T,N> instead.

push_back(T const & value) and emplace_back(T... values)

`push_back(value)` and emplace_back(T... values) both insert a new element 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. insert 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() copies the given value in the new element. With emplace_back() instead, 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, the two operate similarly.

```
std::vector<std::complex<double>> vc;
vc.emplace_back(5.,-3.);// adds 5 - 3i
```

Suggestion: emplace_back() has been added to the language recently and is of more general use than push_back(). So you may decide to forget push_back() and use always emplace_back().

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
```

Testing vector bounds has a cost! Use at() only for debugging or if necessary.

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;
```

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

Resizing and reserving

With resize(n) we change the size of the vector to n (and increase capacity to fit if needed). 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. Element can be addressed with []. Existing element are kept..

reserve(n) instead only deals with the memory buffer and it set the capacity to fit n elements, but the size is unchanged. reserve does not cheage the size of the vector. To add elements we need to use push_back() or emplace_back().

In both cases if n is grater than the current capacity reallocation occurs: iterators and references to elements are invalidated. If n is less than the current capacity capacity is not changed. Existing elements are kept.

Shrinking a vector

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

```
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
```

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
```

Interfacing with C code

The method data() of a std::vector returns the pointer to the memory buffer containing the elements. This is crucial to interfacing with C-code.

```
// a function taking a C array as a pointer
int fun(double * a, int n);
// I my code I use a vector
std::vector<double> a;
// filling the vector vith values
// calling the function (result will be an int)
auto result=fun(a.data(),a,size());
```

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 &)`, `push_front(T const &)` and `emplace_back()`
- Dimensions: `size()` and `capacity()`
- Memory management: `resize(int,T const &=T())`, `reset(int)` `shink_to_size(int)`
- Ranges: `begin()` e `end()`
- Swap with another vector `swap(vector<T> &)`
- Clearing (without releasing memory): `clear()`
- Accessing raw data: `data()`

array<T,N>

std::array<T,N> is a container that encapsulates constant size arrays. It needs the array header file.

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, and memory management.

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

The second template parameter is a value and the corresponding argument should be an integral constant expression containing 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};//an array containing 1,2,3
std::array p{1,2,3};//automatic template ded. Array of 3 ints
std::array<double,2> d = {-3.7,4.8};// copy initial.
d[0]=-1,;// now d contains -1 and 4.8
c.size(); // dimension of array (3)
std::array<std::array<double,3>,3> m; // a simple 3x3 matrix
m[2][0]=-7.8;// setting an element of m
```

Interfacing with C code

Also for standard arrays we have the method data()

```
// a function taking a C array as a pointer
int fun(double * a, int n);
// I my code I use a vector
std::array<double,3> a={-3.,6.7,9.2};
// calling the function (result will be an int)
auto result=fun(a.data(),a.size());
```

Structured binding

Arrays are aggregates, so this construct is possible

```
// a function returning an array
std::array<double,3> fun();
//...
auto [x,y,z]=fun(); //get the returned values
// x,y,z are doubles with values corresponding to
//        the three elements of the returned array.
//another example
std::array<int,3> a={1,2,3};
//...
auto & [l,m,n]=a;// l,m,n are references to elmts of a
l=90; // a is now [90,2,3]
```

This is not possible with vectors;

Traversing vectors and arrays

Here v is a standard vector or array. Traditional way (most flexible):

```
for(std::size_t i=0;i<v.size();++i)
{
  v[i]=10.0; // use of addressing operator
}
```

Using iterators (more uniform)

```
for (auto i=v.begin(); i!=v.end(); ++i)
{
   *i=10.;
}
```

Range based for (remember **auto** & if you want to change the elements of the container):

```
for (auto & i:v)
{
i=10.;
}
```

Iterators

Iterators offer a uniform way to access all Standard containers, in particular vectors and arrays.
Moreover, they are used heavily by standard algorithms. 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 (or any standard container), respectively. You can operate on iterators, for instance $++i$ moves to the iterator to the next element.

Aggregates

Aggregates are just structures with public member data, and member data either of basic type or themselves aggregate. Aggregates enjoy three main properties

- Aggregate initialization;
- Structured binding;
- Easy serialization (so it is easy to transmit an aggregate object in a message). To be technical: an aggregate is trivially copyable.

Let's consider this example of an aggregate

```
struct Aggr{
double a;
int b;
std::array<double,2> c;
};
```

Aggregates

```
Aggr a{10.0,1,{3.,4.}};// aggr. initializ.
auto [x,y,v]=a;// structured binding
// I see the aggregate as a buffer in memory
void * buf=static_cast<void *>(&a);
Aggr aa;// another aggregate
memcpy(static_cast<void *>(&aa),buf,sizeof(Aggr));
// I have copied the buffer, now aa contains the same
// data as a
```

The last part of this example may seem cumbersome: the copy could have been made by simply writing aa=a;. However, the property of trivial serialization of an aggregate may become handy when using Message Passing paradigm. (Besides, memcopy is efficient.)

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;
t={"pippo",3,4,{6.,-3.14}};
// 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.

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 of POD/aggregates 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
```

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.

strings

C++ strings are a light wrapper around C-stype null-terminated strings. They offer however greater safety and easy of use. They require the **#include <string>**.

```
#include <string>
std::string a="Hello-";
std::string b="World";
std::string c=a+b; // c is "Hello World"
// printout:
std::cout<<"the-content-of-c-is-"<<c<<std::endl;
//Extract the underlying C-style string:
auto cc=c.c_str();
// cc is a (C-style) char* containing the null terminated string
// "Hello World
```

Lots of utilities are available for strings, among which tools for regular expressions. See any good reference manual.

`std::string_view`

`std::string_view` is a lightweight, non-owning view over a contiguous character sequence (e.g. a string literal, `std::string`, C array of char). It helps avoids unnecessary copies.

Key properties

- Non-owning: the underlying data must remain alive for the whole lifetime of the view.
- Not required to be null-terminated (unlike C strings).
- Constantness: you cannot modify characters (they are treated as `const char`).
- Ideal for function parameters that accept any "string-like" input without forcing a copy.
- Provides familiar interface: `size()`, `empty()`, `data()`, `operator[]`, `substr()`, `compare()`, `find()`, `starts_with()`, `ends_with()`.

Use of a std::string_view

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

void greet(std::string_view name){
  std::cout << "Hello-" << name << "\n";
}

int main(){
  std::string s = "Alice";
  greet(s);                    // from std::string
  greet("Bob");                // from string literal
  char buffer[] = "Carol";
  greet(buffer);               // from char array
  greet(std::string_view(s).substr(0,3)); // "Ali"
}
```

Pitfalls (lifetime!)

```cpp
// DANGEROUS: temporary destroyed -> dangling view
std::string_view sv = std::string("tmp"); // BAD
// Safe: literal has static storage
std::string_view lit = "persistent";
```

std::pair

The header <utility> introduces pair<T1,T2>, 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;
```