Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Some elements of C++

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

C++ Arrays

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

- Standard arrays derived from C: **double** a[5], **float** c[3][4]. Fixed size
- Dynamic arrays through pointers: **double** *p=**new double**[5]
- The vector container of the standard library: std::vector<**double**> c, which support dynamic memory management: c.resize(100).
- (C++11). Fixed size array. std::array<**double**,5> c

Of course you may have arrays of user defined types (classes): Triangle b[5] in an array of 5 `Triangles`.

Vectors and matrices for numerics

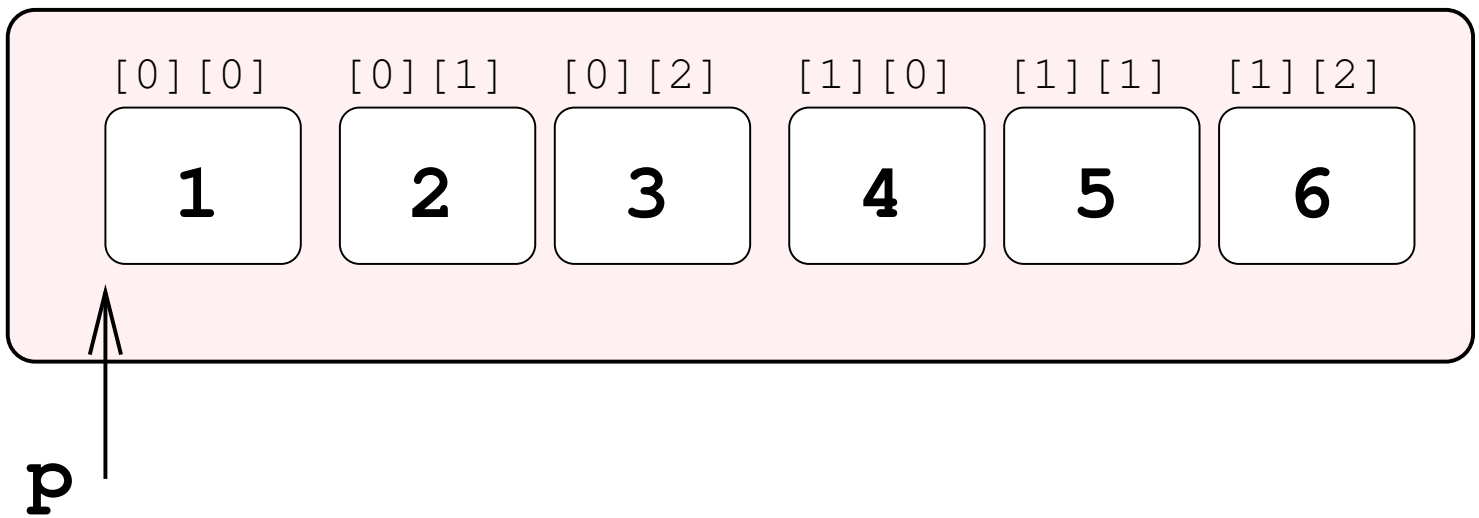
The C++ language does not provide classes for matrices for scientific computing directly, even if the native data structure may form a valid base.
In this course we will use the Eigen vector and matrix classes, highly optimized for SSE 2/3/4, ARM and NEO processors.
We will also give an introduction to the matrices in Trilinos for parallel computing.

However, it is anyway important to understand how standard arrays work. To do that we also make an excursus on pointers, since the concepts are related.

Organization of multidimensional standard arrays

Arrays are organized by row.

```
int p[2][3]={{1,2,3},{4,5,6}}
```



Indefinite dimension

When an array is indicated as formal argument of a function it is possible to omit all dimensions a part the last one.

```
int myfunc(float& a[][2]){
.....
 a[1][1]=5.0; //OK
}
int main(){
float c[10][2];...
int j= myfunc(c);
}
```

This function take as a argument a reference to a **double** [][2], i.e. a bidimensional array whose second dimension is 2.

Pointers

Pointers are able to store addresses of an object. They are mainly used or to hold dynamically a resource (i.e. an object that we want to use in our program), to implement polymorphism, or to hold a standard dynamic array.

We will analyze all those situations in our examples and we will introduce also the new smart pointers of C++11.

The use of the containers of the standard library (like vector<>) reduces to a large extent the use of pointers for dynamically allocated arrays, thus obtaining a safer code.

Yet, for completeness we recall how pointer and arrays are related.

Example of use of pointers

```
int* pi(0) ; // pointer to a int whose value is 0
char** ppc ; // pointer to a pointer of char
int* ap[15]; // array of 15 pointers to int
double (*) v[3];// a pointer to an array of 3 doubles
int (*fp)(char*); // pointer to a function
// that takes a char* as argument and returns a int
int f; pi=&f';//now pi points to f
(*pi == f) ;// the result is true
```

The value 0 or (C++11) the keyword nullptr indicates a null pointer. The prefix operator * dereference a pointer, returning the "pointed value". The prefix operator (&) returns the address of an object.

Pointers and C-style dynamic arrays

The two concept are related. Why? It is simple. For a pointer p, p[3] is equivalent to $*(p+3)$. The addition of 3 to a pointer moves the pointer forward of $3*$**sizeof**$(*p)$ bytes.

```
double * pa;
pa=new double[10];//pa points to an array of 10 double
pa[3]=9.0;
...
delete []pa;
```

Operator `new` requests memory to the operative system and returns a pointer. It has three forms

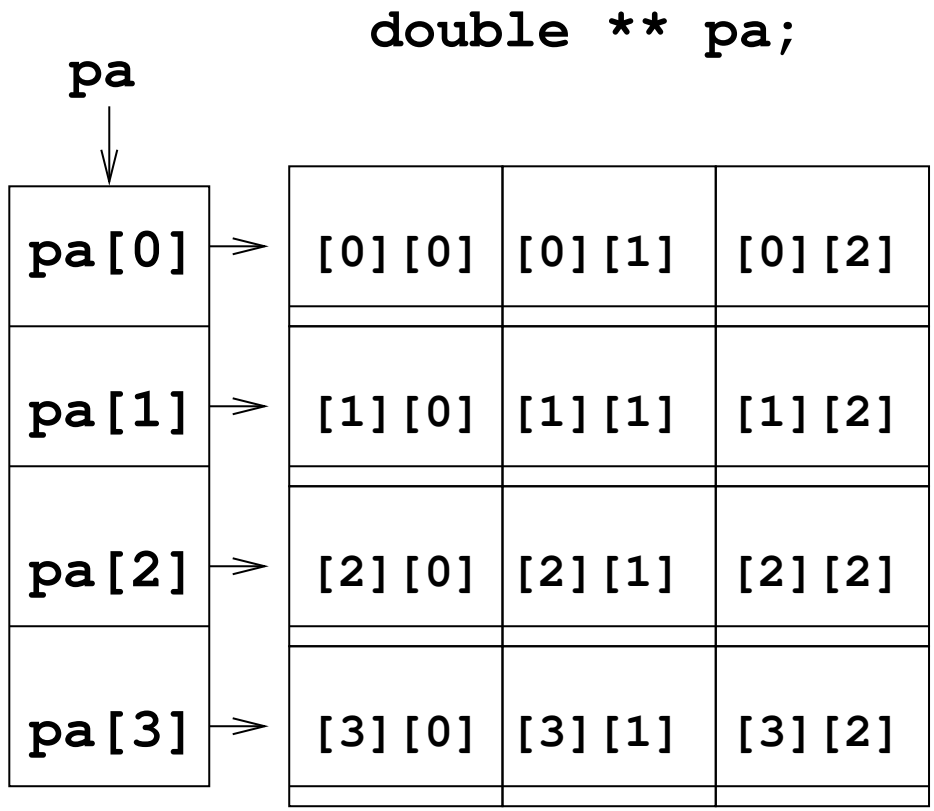
`T* new T`	Creates an object of type `T`
`T* new T(z)`	Creates an object of type `T` using the constructor with argument `z`
`T* new T[m]`	Creates an array of m elements of type `T`

Double pointers and multidimensional arrays

```
double ** pa;
pa= new double *[4];
for(int i=0;i<4;++i)pa[i]=new double[3](3.0);
// I can address pa as a double[4][3];
for(int i=0;i<4;++i) for(int j=0;j<3;++j)pa[i][j]=i*j;
..
for(int i=0;i<4;++i) delete[] pa[i];
delete[] pa;
```

Remember that every **new** should be matched by a **delete** and **new[]** by **delete[]**. It is because of the risks of making mistakes in memory management that in C++ one avoids to use pointers for handling dynamic arrays and uses more specialized structures, like stl containers.

Memory layout



pa[i][j] translates into $*(*(pa+i)+j)$ which offsets pointer pa of i∗**sizeof**(**double** ∗). It follows an offset of j∗**sizeof**(**double**) to reach the wanted address.

Some common errors

```
double *p; double * t;
p=new double[10];
delete p;// delete []!
delete t;// t has not been assigned!
```

We have two fatal errors. The second would not be an error if we had initialized `t` with the null pointer!
The first error is trickier. Having called **delete** and not **delete**[] we are destroying only the first element of the array! We have created a memory leak.

General rules: Reduce the use of bare pointers for dynamic arrays, prefer stl containers. Always initialize a pointer to a valid address or to the null pointer. Or, use smart pointers!

Some common errors

```
double *p; double * t;
p=new double[10];
delete p;// delete []!
delete t;// t has not been assigned!
```

We have two fatal errors. The second would not be an error if we had initialized `t` with the null pointer!
The first error is trickier. Having called **delete** and not **delete**[] we are destroying only the first element of the array! We have created a memory leak.

General rules: Reduce the use of bare pointers for dynamic arrays, prefer stl containers. Always initialize a pointer to a valid address or to the null pointer. Or, use smart pointers!

Efficiency issues

Multiple pointer for dynamic multidimensional arrays are not very efficient since the memory layout is, in general, only partially contiguous. Optimization of cache access is thus hindered. to hinder = ostacolare

For multidimensional arrays it is better to use specialized libraries, like the `Eigen` or to develop special classes. Normally these libraries implement multidimensional arrays using linear contiguous memory.

The only case where C++ native multidimensional arrays do not introduce a reduction in efficiency is when just one dimension is dynamic. We present hera an implementation using vector and array classes (more C++ style).

Arrays with one dynamic dimension

Lets consider, for instance, an two dimensional array which contains the coordinates of a 3D grid. We will know the number of points only at run-time, but we already know the number of coordinates for each point, in this case 3.

A pointer to an array whose second dimension is 3 is defined as follows:

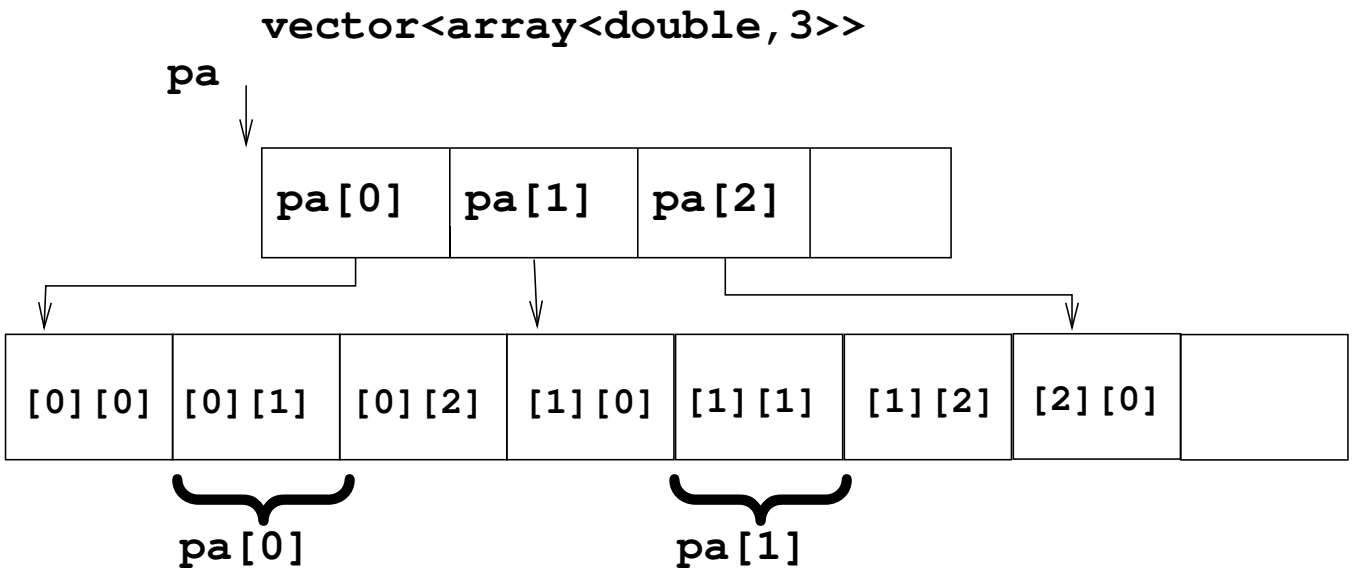
```
double (*) v[3];
```

But, alas, we have C++11 now, and life is easier if we use instead

```
std::vector<std::array<double,3>> v;
```

Arrays with one dynamic dimension (C++11)

```
using AD=std::vector<std::array<double,3>>; //for simplicity
... // read n
AD pa(n); // a vector of n arrays
//I fill the array with values
for (int i=0;i<n;++i)
   for (int j=0; j<3; ++j) pa[i][j]=double(i*j);
```



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 principal one is `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 in the namespace `std`.

Computational complexity of main operations on vector<>

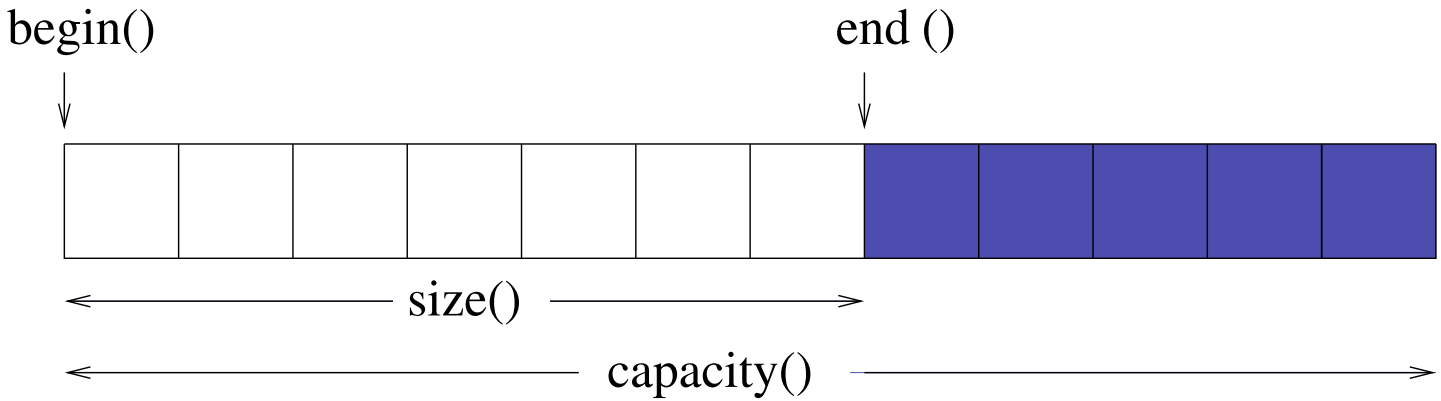
Random access	O(1)
Adding/deleting element to end	O(1)[1]
Adding/deleting arbitrary position	O(N)

Note: we sometimes 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

La dimensione è nota al runtime, non è neccesario sia nota al compile time
E' possibile modificare la lunghezza

Internal organization



In a vector data is stored in a dynamically allocated contiguous memory area. The size of this area, in terms of number of elements that can be stored, is the *capacity*. The *size* of a vector refers to the portion of this area actually filled with vector elements. The capacity is automatically increased if insufficient when using methods that insert new elements. It is never decreased automatically.

vector<T>, like all containers, may be accessed through iterators (a sort of pointers). Methods begin() and end() return the *iterator* to the *first* and *last element+1*.

Examples

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

Both *size* and *capacity* is 0.

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

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

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

The elements are constructed using `float(3.14)` (copy constructor).

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.

- If *size*+1>*capacity*
  a. allocate a larger capacity (usually doubles the current one) and correct *capacity* accordingly;
  b. copy current elements in the new memory area;
  c. free the old memory area;

  Add the new element at the end of the vector and set *size*=*size*+1;

emplace_back(T... values)

C++11 has introduced a new feature, that avoids the need of making copies or using the default constructor when adding elements to an array. It employs variadic template parameters (we will talk about this topic in another lecture). It is sufficient to know that with emplace_back we may directly pass arguments to the constructor, so the stored object is directly constructed in memory, with computational savings!.
In practice, it operates similarly to push_back, so it adds a new element at the back of the vector.

```
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 taked 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 a cannot use push_back(value) with value an object of type MyClass.
While, I can use emplace_back() !!

And if there is not enough memory?

If a push_back() oremplace_back() fails because we have run out of memory, an exception of type std::bad_alloc is thrown (`bad_alloc`) is derived from `std::exception`) . If not captured the program aborts with an error message.
If it is captured, the standard ensure that the vector is in a valid state equal to the one before the addition of the element.

```
#include <exception>
try{
  a.push_back(10); //ad an element
}
catch (std::bad_alloc & e){
  // memory insufficient
  // a is still a valid vector
}
```

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 extensive! Use it only for debugging!

What 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, particularly when dealing with very large vectors on 64 bit architectures, it is better not to guess. A vector<T> knows the type used to address its element. It is a typedef 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){
  ....
 }
```

It is longer but safer. Alternative: use iterators, or the new container addressing introduced by C++11 (shown later) or use std::algorithm (we will see them in a next lecture)

Reservation, please

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

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

vector<float>d;
d.resize(1000);\\ resizes the vector
// and fills it with values using float()
for (i=0;i<1000,++i)d[i]=i*i;
```

The second technique is the most efficient. The first is the least efficient because of the repeated memory allocations/deallocations. The second is intermediate.

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 copy constructor is used.

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() (C++11).

Note: Use reserve() whenever possible: you get a more efficient code.

Con resize: se la nuova dimensione è superiore a quella vecchia, i vecchi elementi rimangono inalterati, quelli nuovi vengono inizializzati con default/copy constructor
Se la nuova dimensione è inferiore a quella vecchia: cambia la size ma i valori rimangono inalterati --> il secondo argomento di resize è ininfluente

Shrinking a vector

Sometimes it may be useful to shrink the capacity of a vector to its actual size. Unfortunately C++98 does not provide a method for that. But there is a trick which uses the method swap() (which swaps the content of two vectors).

```
vector<double>a;
...// I do something with the vector
// Now I want to shrink it
vector<double>(a).swap(a)
```

I create a temporary vector copying a vector<**double**>(a)) and then I swap it with a.

In C++11 we have a new special method for this purpose:

```
a.shrink_to_fit() // At last!
```

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 (vector<double>::iterator 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.

In C++11 we may use auto:

```
for (auto i=a.begin(); i!=a.end(); ++i)*i=10.56;
```

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: `a[i]` $\rightarrow$ `*(&a[0] + i)`, while de-referencing an iterator is faster (but methods `begin()` and `end()` have a cost...) ;
- 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 (C++11 only)

In C++11 there is 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;
```

In C++11 **auto** i : Container generates a variable (i) that will hold the value of the elements in succession. Beware: if you want to change the values you need to use **auto** & i : Container. This way you obtain a reference to the elements of the container (and not just their value).

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().

```
it=a.begin();
v=a[5];// OK
a[2]=-7.6;// OK
a.push_back(7.8); //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!!!
```

Beware that a constant vector may be iterated only using constant iterators.

Methods cbegin() and cend() are equivalent to begin() and end() but return constant iterators (since C++11). So I could have written the previous statement as

```
auto b = a.cbegin();
```

Going reverse

We can use iterators to address 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++ de-references iterator j and "advances" it (dereferencing operator has precedence over the post-increment operator). But, being j a reverse iterator, advancing ... means retreating!

Pointers and vector<T>

Sometimes may be useful to access directly the memory area of a vector<> through a pointer. In C++98 this is possible using this trick.

```
double myf(double const * x, int dim);//requires double*
...
vector<double> r;
...
y=myf(&r[0],r.size());
```

In C++11 life is easier, since we have the method data() just for this purpose:

```
y=myf(r.data(),r.size());
```

Restituisce un puntatore all'array utilizzato internamente dalla classe vector per immagazzinare i dati

Note: this technique is allowed only if you do not allocate/deallocate data using the pointer! Statements like r.data()=**new double**[100] are obviously FORBIDDEN!.

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)` (only C++11)
- Ranges `begin()` e `end()`
- Swap `swap(vector<T> &)`
- Clearing (without releasing memory): `clear()`
- Accessing data: `data()` (C++11 only)

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) (C++11 only)
- `vector<T>::reference` (`vector<T>::const_reference`) Reference to elements (const variant) (C++11 only)

array<T,N>

std::array<T,N> is a container that encapsulates constant size arrays.
It has the same semantics as a C-style array. 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 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

```
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
c.size(); // dimension of array (3)
std::sort(a.begin(), a.end());// iterators to an array
for(auto s: a) std::cout << s << ' '; //Range based for
```

array<> has been introduced to have a replacement of C-style array compliant with stl containers and with the same efficiency.

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

Testing the size of an array at compile time

The method size() computes the size of the array at run time.
Sometimes, for efficiency, we want it to be known at compile time.
We need to use a special static member of tuple_size<T>.

```
#include <iostream>
#include <array>

template<class T>
void test(T t){
 int a[std::tuple_size<T>::value];
 std::cout << std::tuple_size<T>::value << '\n'; }

int main(){
   std::array<float, 3> arr;
   test(arr);
}
```

The result is 3. tuple_size<T>::value is evaluated at compile time/
Of course tuple_size<> can operate only on array<> and not on vector<>

Why Pointers?

Pointers are used for different reasons

- To watch (point to) a resource whose lifespan is independent (but should exceed) that of the pointer;
- To held a resource whose lifespan should coincide with that of the pointer. It's an owing pointer: it has a unique ownership of the object; --> unique pointers
- To address a resource that is shared with other pointers and should be destroyed when the last pointer goes out of scope. In this case we have a shared ownership. --> shared, weak pointers

For all but the first issue you'd better use smart pointers

Smart pointers and C++

With C++98 the language had a very limited support to smart pointers. The only smart pointer was `auto_prt`, which was poorly designed and indeed it is now deprecated. C++11 has integrated (and bettered) smart pointers previously provided by the Boost libraries. 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.

They all require the `<memory>` header.

The use of unique_ptr

Let's look at an example

```
enum Shape{Triangle,Square};
class myClass{
setPolygon(Polygon * p){my_polygon=p;}
...
private:
Polygon * my_polygon;
}
// A Factory of Polygons
Polygon * polyFactory(Shape t){
switch(t){
case Triangle: return new Triangle;
case Square:     return new Square;
..
default: return nullprt;
}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

A poor design

This design is prone to error. First of all the object of type MyClass has now to take care of handling of the resource my_polygon. We need to build constructors, destructor, assignment operator very carefully, and 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 required, causing a memory leak which may be difficult to detect!

The version with unique_ptr

```
class myClass{
setPolygon(unique_ptr<Polygon> p){my_polygon=move(p);}
...
private:
unique_ptr<Polygon> my_polygon;
}
// A Factory of Polygons
unique_ptr<Polygon> polyFactory(Shape t){
switch(t){
case Triangle: return unique_ptr<Polygon>(new Triangle);
case Square:     return unique_ptr<Polygon>(new Square);
..
default: return unique_ptr<Polygon>();
}
...
MyClass a; a.setPolygon(polyFactory("Triangle"));
```

Complete example in SmartPointers

How a unique_ptr<> works

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!
```

The default constructor produces an empty unique pointer. You may check if a unique_ptr is empty by testing prt == nullprt. In C++14 we may build a unique pointer via the `make_unique(...)` function. To create a shared pointer initialized with Triangle():

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

Moving unique_ptrs around

Unique pointer implement the move semantic (there will be a lecture on move semantic later on!). In particular, if initialized with a *temporary object* of unique_ptr type the ownership of the resource is transferred (this is what happens with the value returned by polyFactory in the example).

It is not assignable but ownership can be transferred using the std::move utility:

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

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

Dealing with 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]);// IS A SERIOUS ERROR!!
unique_ptr<string[]> up(new string[10]);// OK
```

The second version calls **delete**[] when the pointer gets out of scope. This is a big improvement with respect to the old auto_prt<T>.

Main methods and utilities of unique_ptr

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.
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.

unique_ptrs can be stored in a standard container:

```
vector<unique_ptr<AbstractPolygon>> polygons;
```

This is another big change w.r.t. auto_ptr.

Shared pointers

Assume you have to "refer" in several places of your program 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 "cleanup when the object is nowhere used anymore".

However, shared pointers are rather heavy (particularly in multi-thread environments), so use them only if really needed. If the ownership is in fact unique, use a unique_ptr instead (which is much lighter).
See the example in SmartPointers

References

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

```
double horner(double & x);
//???
double & pippo(int i)double a;...;return a;
double & c= pippo(3);// OK
double& a=horner(5)//NO!;
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!
```

Reference semantic in std containers

Std containers can hold or first class objects or (normal) pointers. Smart pointer class can also be stored in a container, but not references:

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

Reference semantic in std containers

However, C++11 has introduced a way to store references in a container. This can be useful, as an alternative to pointers. 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);// it stores a reference
v.push_back(p2);//
p1.setCoord(7,8);// change coordinates of p1
```

Also v[0] has changed coordinates since it is a reference!.

Constant variables

The type qualifier **const** indicates that the object cannot be changed. A constant variable must then be initialized to a value. Constant class member variables can be initialized in the class declaration only if they are of integral type. This is not true for constexpr.

```
double const pi=3.14159265358979;
float const e(2.7182818f);
double const Pi=std::atan(1.0)*4.0;
const unsigned ndim=3u;
```

In C++11 it is better to use constexpr (when possible)

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

Constant class member variables

```
class MyClass{
 ...
 static constexpr int ndim=2; //OK
 constexpr double Pi=3.1415;//OK
// const double Pi=3.1415;//ERROR
 const double pi;// OK!!
};
...

const MyClass::pi=std::atan(1.0)*4.0; //OK!
```

Inizialization of a constant member variable is usually put in the header file, as well as that of global constants.

constance rules

`const` is associated to the type on its left. unless it is the first keyword of the statement. In the latter case it applies to the type on its right.

```
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!.

Note: A reference is always const (reference cannot be reassigned). So **double** & **const** a is simply wrong!

const (and constexpr) is your friend!

Use them!. Particularly in the declaration of function arguments when the arguments are references:

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

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

- Your program is safer and easier to understand. If you mean that a variable should not be changed, with **const** you make it clear to the world.
- You avoid to modify by mistake a variable that was meant to hold a constant value.
- You get a more efficient code: the compiler is able to make on constant values optimization that cannot be made otherwise!! In particular, constant expressions are computed at compile time.

literal and const expressions

A literal expression is a string that represents a value. In

```
double constexpr pi=3.14159265358979;
float constexpr e(2.7182818f);
```

the strings `3.14159265358979` and `2.7182818f` represent a double and a float value, respectively. Obviously literal expressions are constants! I may have used just `const` in the example. `constexpr` however in this case is better (C++11)!
A constant expression is an expression that may be computed at compile time. C++11 allows to specify that a function may return such type of expressions using the keyword `constexpr`:

```
constexpr double cube(double x)
{ return x*x*x;} // ONLY C++11
```

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

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

constexpr

The new keyword `constexpr` may help develop faster code and ease some *metaprogramming techniques*. But has STRONG limitations. In particular, a function returning a `constexpr` value has to comply with the following restrictions

- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

C++14 has somehow relaxed these requirements!!!

const_cast<T>

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

```
double minmod(float & a, float & b);
...
double fluxlimit(float const& ul,float const& ur)
...
l=minmod(const_cast<float>(ul),const_cast<float>(ur));
...
```

You cannot const_case a constant expression!.

mutable members

If a class object is declared const, you can only access its member, not modify them. More precisely, only const methods you access const members. In a few particular cases it may be necessary to say that some members can modified even on constant objects. You may use **mutable**.

```
struct foo{
  double a;
  mutable int n;
}
....
const foo s{3.0,5}// a=3.0, n=5
s.a=6.0; // ERROR s is const!
s.n=9;// OK!
```

Enumerations

I assume that you already now about enumerations:

```
enum bctype{Dirichlet,Neumann,Robin};// definizione
...
bctype bc=Neumann;// ..
switch(bc){
case Dirichlet:
...
break;
case Neumann:
...
break;
Default:
...
}
```

Class enumerators

In C++ enumerators are in fact integral type, always convertible to integers. C++11 has introduces strog typed enumerators which behave as a user defined type and are not convertible to int. This may allow to write a safer code. You just have to ass **class** to the declaration:

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

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

Now the test **if** (color==0) would fail.

Functions

I recall the following concepts.

- Declaration of a function. its role is to provide the compiler with the signature of the function (name + type of arguments) and its return type. In a pure declaration (i.e. a declaration that is not a definition) you can omit the name of the arguments: **double** foo(**double** &, **int**). It is normally contained in the header file.
- Definition of a function Provides the (*body*) of the function. A definition is also a declaration. Only one definition can occur in a program.

```
double cylVolume(const double,const double);
double polyArea(const vector<lati> &);
```

Passing arguments by value or by reference

```
int fun1(const int i, float b, double c);
float fun2(int& i, float& b, double const & c);
...
int s=fun1(5, z, r);
float g=fun2(h, z, 3.5);
```

In `fun1` the arguments `i`, `b` and `c` are passed by value. It means that they correspond to local variables and modifications in `fun1` will not alter the value of the actual arguments in the calling code. Variable `i` is declared *const*. It means that it will not be modified inside `fun1`. Having passed this argument by value this is irrelevant for the calling program.

```
float fun2(int& i, float& b, double const & c);
```

In `fun2` arguments are passed by reference. It means that they will be references to the corresponding variables in the calling code. A modification of `i` inside `fun2` will reflect on the corresponding actual argument

Using references, no copy of the value is made inside the function (memory saving!!).

Note the use of *const*: `c` cannot be changed. Therefore it is just an input. When passing by reference the use of the `const` keyword for input arguments is important!

Passing pointers

Of course it is possible to use pointers, with a similar effect than passing references.

```
void LU(Matrix const * A, Matrix * L, Matrix * U);
```

Here `A` is a pointer to an argument of type `Matrix` which is just an input, it will not be modified by the function: it is indeed a pointer to a constant. The values held by the other pointers may (and in general will) be modified.
Personally, unless there is a special reason to use pointers, I prefer using references.
Note: Passing pointers or references is indeed another way of receiving a "*return value*" from a function.

General guidelines

- Prefer passing by reference when dealing with large data. You avoid to copy the data in the function local variable.
- Always declare const input variables (i.e. variables whose value is used but not changed inside the function) passed by reference or by pointers.
- A literal constant may be given as actual argument only if passed by value or by constant reference (`const &`).
- Passing non constant references does not allow the recursive use of the function (I suggest to avoid recursive function calls anyway, it is usually inefficient).

Can you find the errors?

```
void pList(Mesh const & mesh,int * list, int const bc){
unsigned count=0;
for(int i=0;i<mesh.numnodes();++i)
if(mesh.nodeBc(i)==bc)++count;
list=new int[count];
...
}
...
int mylist;
pList(mymesh,&mylist,5);
```

There is a fatal error!.

Can you find the errors?

```
int * pList(Mesh const & mesh, int const bc){
unsigned count=0;
for(int i=0;i<mesh.numnodes();++i)
if(mesh.nodeBc(i)==bc)++count;
int * list=new int[count];
...
return list; }
...
int * mylist=pList(mymesh,mylist,5);
```

`list` list must be returned (or you may use a reference to a pointer)

Can you find the errors?

```
void pList(mesh const & mesh, vector<int> & list,
int const bc){
unsigned count=0;
for(int i=0;i<mesh.numnodes();++i)
if(mesh.nodeBc(i)==bc)++count;
list.resize(count);
...
}
...
vector<int> mylist;
pList(mymesh,mylist,5);
```

Still better: use a `vector<T>`

Function overloading

In C++ (different from C!) the identifier of a function is not is name, but its signature, which is formed by the name and the type of the arguments.
Thus two functions with the same name but different argument types are considered as two different items. This is the basic mechanism that allows function overloading.

```
float fun(const float * & a, vector<float> & b);
double fun(const double * & a, vector<double> & b);
void fun(int & a);
...
double * z, vector<double> x, int l;
double g=fun(z,x);
//calls float fun(const double * &, vector<double> & b)
...
fun(l);//calls fun(int & a)
```

The compiler selects the function whose *signature* best matches the call, taking into account also implicit conversions.

Default arguments

In the declaration of a function, the rightmost arguments 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
...
```

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;\newline
if(first){
// Executed only the first time
first=false;}else{
 //Executed from the second call onwards
...
}
...
```

Pointer to functions

```
double integrand(const double & x);
...
typedef double (*pf)(const double &);
double simpson(const double a, const double b, pf const f, const
...

integral= simpson(0,3.1415, integrand,150);

// Using a pointer to function
pf p_int=integrand;
integral= simpson(0,3.1415, p_int,150);
...
```

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

A note

It may be preferable to avoid pointers to functions. C++ allows several way of avoiding it: object function (also called functors), function wrappers and lambda function (the latter two only with C++11). We will have a lecture dedicated to that.

New function declaration syntax

C++11 has introduced a new function declaration syntax

```
// new syntax for double fun(double, int)
auto fun(double, int) -> double;
```

The reason is that sometimes the return type of a function depends on an expression processed with the arguments

```
template<class T1, class T2>
auto add (T1 x, T2 y) -> decltype(x+y);
```

The return type is defined as the type of the result of $x + y$. Using the more common syntax

```
template<class T1, class T2>
 decltype(x+y) add (T1 x, T2 y);
```

is an error since `x` and `y` are not in scope when `decltype` is used.

An example

In the directory Horner you find an example that used 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. But, more importantly, check how std::pow() is a performance killer.

New fixed width integer types (C++11)

With the proliferation of different architecture to write safe code sometimes it may be necessary to specify exactly what we mean with "integer": is a 16 bit integer, a 32 bit? The header li!¡cstdint¿! in c++11 defines new 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: 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!