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

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

C++ Arrays and Vectors

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

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

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

`std::vector<T>`

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

it is hence cache-friendly for the sequential access

Computational complexity of main operations on vector$<>$

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

so on the small scale it excels, wins over other in-theory-better structure

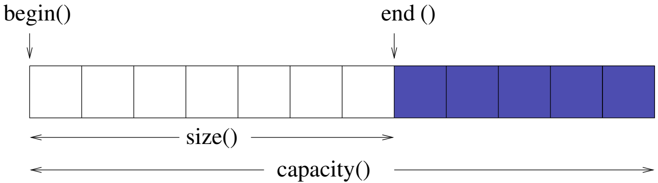
Note: we normally write vector$<$T$>$ to remember that standard vectors have a compulsory template argument. But in fact the template arguments are 2!. The second is the allocator, which has a default value and is rarely changed.

[1] If the capacity is sufficient

The structure of a vector

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

le "dimensioni" sono intese come numero di valori riempi-bili, non byte o altro



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

Examples of `vector<>`

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

Both *size* and *capacity* is 0.

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

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

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

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

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

Size is 2 and capacity at least 2.

Automatic deduction of template parameters

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

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

This is indeed a nice simplification for short vectors.

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

push_back(T const & value)

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

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

emplace_back(T... values)

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

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

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

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

Usage of emplace_back

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

morally maybe emplace-back will use properly the ctors available

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

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

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

Addressing elements of a vector<T>

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

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

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

Which is the type of an index?

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

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

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

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

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

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

Reservation, please

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

vector<float>c;
```

this does not change the size

```
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
```

this does not change the capacity

```
// 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 because they avoid memory allocations/deallocations.

Resizing and reserving

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

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

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

Shrinking a vector

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

we it still occupies memory

```
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
```

Iterators

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

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

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

Iterators versus classical loops

We could have operated in a more classical way

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

Using iterators may have some advantages:

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

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

Range based for-loops

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

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

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

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

since auto strips all the qualifiers

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

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

An important note

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

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

const_iterator

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

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

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

Going reverse

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

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

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

Interfacing with legacy code

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

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

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

Main methods of `vector<T>`

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

Main types defined by `vector<T>`

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

array<T,N>

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

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

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

The second template argument is the size of the array

examples of standard arrays

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

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

Structured Bindings

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

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

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

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

Testing the size of an array at compile time

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

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

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

Tuples

Tuples are fixed size collections of object of different types.

```
#include <tuple>
```

best way of creating a simple struct/class

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

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

Suggestion: always use make_tuple to create a tuple.

Extracting/changing elements of a tuple

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

- Use the utility std::get<>

as a reference, so to R and W elem

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

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

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

here we are creating these objects (s, i, x, ecc), while w/ these already exist we use tie

Extracting/changing elements of a tuple

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

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

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

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

With tie you can also assign values to a tuple

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

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

pair

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

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

Suggestion: always use make_pair to create a pair.

Vectors and matrices for numerics

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

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

We will make a special lecture on that.