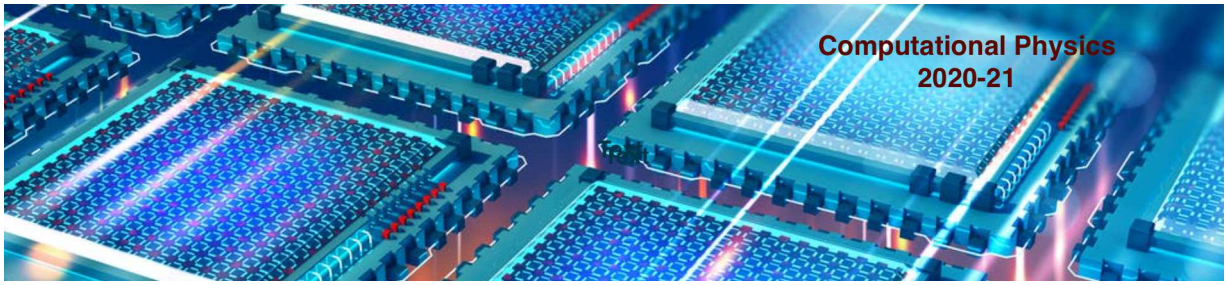
Computational Physics

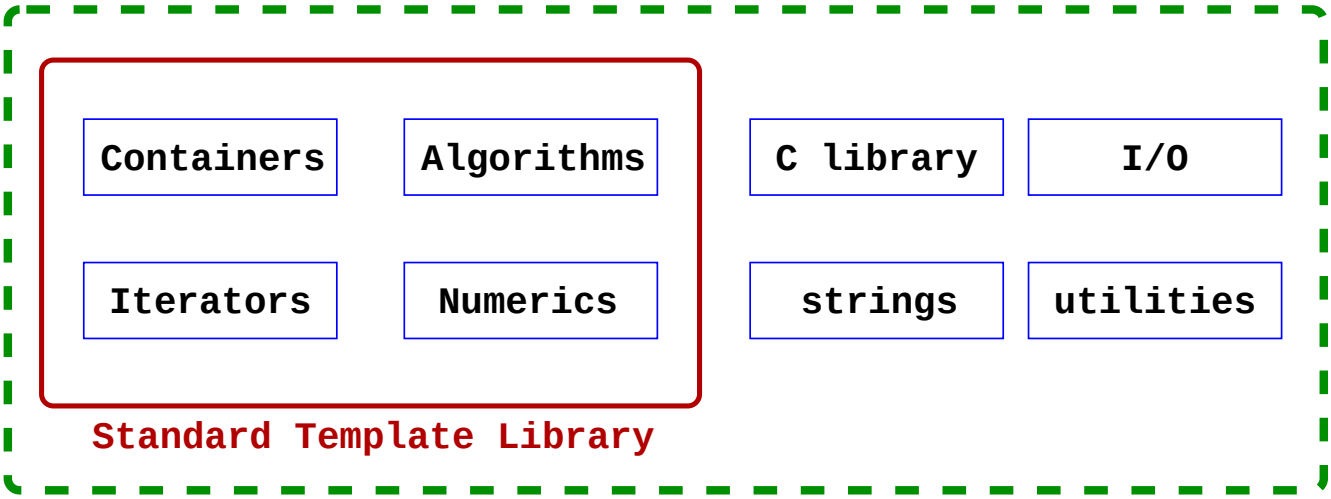
numerical methods with C++ (and UNIX)

2020-21



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: fernando.barao@tecnico.ulisboa.pt

C++ standard library



The C++ *STL (Standard Template Library)* is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like *vectors, lists, queues, and stacks*

C++ STL library: containers

- **Containers**
  A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates (not treated in this course), which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through *iterators* (reference objects with similar properties to pointers).
- Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...
- Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity).
- stack, queue and priority_queue are implemented as container adapters. Container adapters are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as *deque* or *list*) to handle the elements.

C++ STL library: containers (cont.)

- **Sequences**
  - vector: Dynamic array of variables, struct or objects. Insert data at the end.
  - deque: Array which supports insertion/removal of elements at beginning or end of array
  - list: Linked list of variables, struct or objects. Insert/remove anywhere.
- **Associative Containers**
  - set (duplicate data not allowed in set), multiset (duplication allowed) Collection of ordered data in a balanced binary tree structure. Fast search.
  - map (unique keys), multimap (duplicate keys allowed) Associative key-value pair held in balanced binary tree structure.
- **Container adapters**
  - stack LIFO
  - queue FIFO
  - priority_queue returns element with highest priority.
- **Operations/Utilities**
  - iterator STL class to represent position in an STL container. An iterator is declared to be associated with a single container class type.
  - algorithm Routines to find, count, sort, search, ... elements in container classes

C++ STL library

```
Sequence containers:
 - array                          |  ...  |  | <---- LIFO
 - vector                         |  3rd  |  |
 - deque                          |  2nd  |  |
 - forward_list                   |__1st__| \/ pushing elements
 - list

Container adaptors:
 - stack: LIFO stack (class template )
 - queue: FIFO queue (class template )
 - priority_queue

Associative containers:
 - set
 - multiset
 - map
 - multimap
```


C++ STL library: iterators

- ✔ Iterators allow us to move on the elements of containers
- ✔ STL algorithms use them to access containers
- ✔ Iterators elements are in the header *<iterator>*
- ✔ Container and algorithm headers have iterator header included (you don't need to include it explicitely)
- ✔ Every container defines two iterator types:
  container::iterator
  is provided to iterate over elements in read/write mode

```
list<char>::iterator pos;
```

  container::const _iterator
  is provided to iterate over elements in read-only mode

```
list<char>::const_iterator pos;
```

C++ STL library: vector container

vector container

similar to an array but can be dynamically enlarged or shrinked

create and fill vector

```
#include <iostream>
#include <vector>
#include <algorithm> // sort vector
using namespace std;

int main() {
   vector<float> vec; // create a vector to store floats

   // push 5 random values between 0 and 1 into the vector
   for (int i = 0; i < 5; i++) {
      float f = rand()/(float)RAND_MAX;
      vec.push_back(f);
    }
   // vector size
   cout << "vector size=" << vec.size() << endl;
```


C++ STL library: vector (cont.)

accessing vector elements: operator [] and iterator

```
   // add 5 vector values
   float sum = 0;
   for(int i = 0; i < 5; i++){
      sum += vec[i]; // vec.at(i) could also be used
   }

   // use iterator to access the values
   vector<int>::iterator vecit = vec.begin();
   while( vecit != vec.end()) {
      cout << "value =" << *vectit << endl;
      vecit++;
   }
```

fill vector from 1D-array

```
   // fill vector
   int myints[] = {32,71,12,45,26,80,53,33};
   vector<int> v(myints, myints+8); // 32 71 12 45 26 80 53 33

   // clear vector
   v.clear();
```

C++ STL library: vector (cont.)

creating vectors

```
// an empty vector of integers
vector <int> v;

// a vector with 5 elements,
// each an integer
vector<int> v1(5);

// An array of 5 empty
// vector<int> elements
vector<int> va[5];

// A vector with 5 elements each
// having the value 15
vector<int> v2(5, 15);

// A vector with size and values
// of other vector v2
vector<int> v3(v2);

// A vector with same contents of v2
vector<int> v4(v2.begin(),v2.end());
```

creating vectors

```
// Create a vector from an array
// and store first 4 values
int a[] = {1,2,3,4,5,6};
vector<int> v5(&a[0], &a[0]+4);

// assign(InpIter first, InpIter last)
// the range used [first, last]:
// all elements
// between first and last,
// but do not including last
vector<int> v5;
v5.assign(a, a+4); // 1,2,3,4
```


C++ STL library: vector (cont.)

creating matrices: vector of vectors

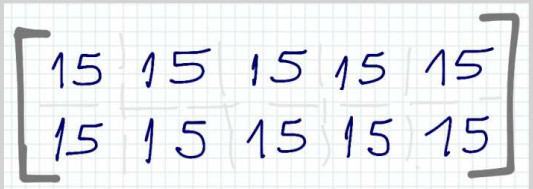
```
// An empty vector of vectors.
// The space appearing between the 2 end greater signs is mandatory
  vector<vector<int> > v2d;

// If you intend creating many vector of vectors
  typedef vector<vector<int> > vecM;
  vecM matrix;

// Create a 2 x 5 matrix
// ...First, create a row vector (5 elements)
  vector<int> vr(5, 15);
// ...Now create a vector of 2 elements with each element a copy of vr
  vector<vector<int> > vm(2,vr);

// Print out the elements
  for(int i=0;i<vm.size(); i++) { //loop on rows
    for (int j=0;j<vm[i].size(); j++) {// loop on every row elem
        cout << vm[i][j] << " ";}
    cout << endl;
  }

  //clean
  vm.clear();
```

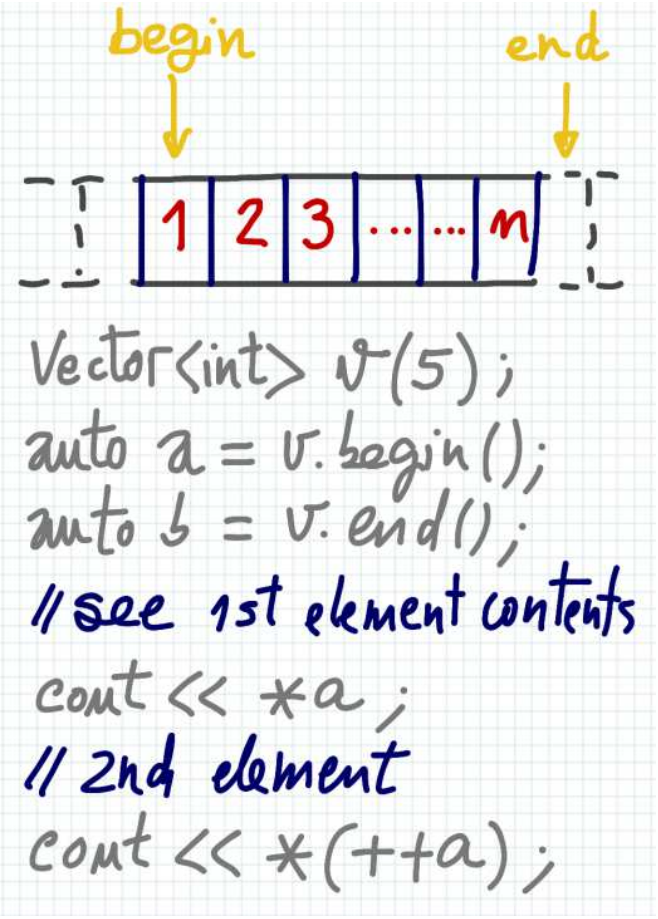


C++ STL library: iterator operations

- ✔ Read and/or write to the element it is pointing to by using dereferencing operators * or ->
- ✔ Go to next element by using increment operator ++

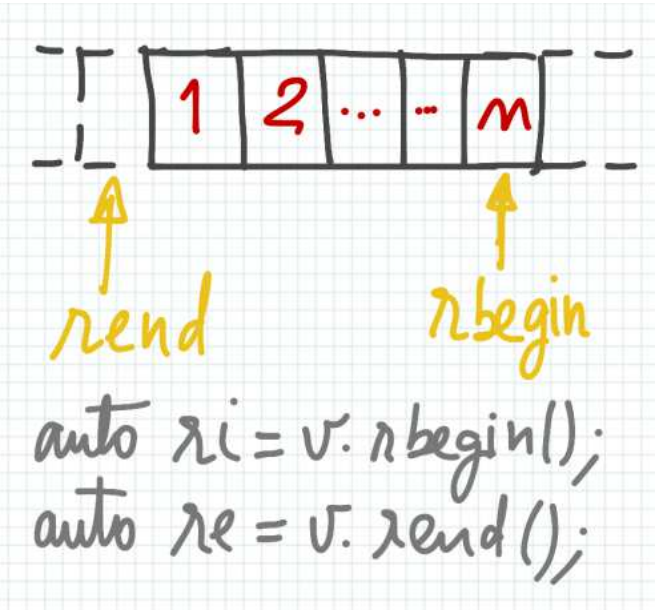
```
++a // step forward one element,
    // return new position
a++ // step forward one element,
    // return old position
```

- ✔ Check if it is equal to another iterator using operator ==



C++ STL library: iterator operations

- ✔ Reverse iterator allows to traverse container elements from end to begin
- ✔ Using the increment operator ++ we move backward and the decrement operator - - moves forward



C++ STL library: pair

pair

This class couples together a pair of values, which may be of different types. The individual values can be accessed through its public members **first** and **second**.

creating pair objects

```
#include <utility>      // std::pair, std::make_pair
#include <string>       // std::string
#include <iostream>     // std::cout

(...)
// declare pair variables and construct object contents
std::pair <std::string,double> planet1, planet2("Mars",3389.5 );
planet1 = std::make_pair("Earth",6371);

// make a copy
vector<std::string,  double> homeplanet = planet1; // = operator working!
```


C++ STL library: pair

pair accessors: first, second

```
(...)

// access pair elements
std::cout << "Home planet: " << homeplanet.first << '\n';
std::cout << "Planet size: " << homeplanet.second << '\n';

// build pair with accessors
vector<std::string,  double> planet3;
planet3.first = "Mercury";
planet3.second = 2439.7;
```

vector of pair objects

```
(...)

// vector of pairs
vector<pair<int,int>> vpair;
vpair.push_back(std::make_pair(1,2));
vpair.push_back(std::make_pair(3,4));
```

C++ STL library: list

list

Compared to other base standard sequence containers (array, vector and deque), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained.

The main drawback of lists compared to these other sequence containers is that they lack direct access to the elements by their position: to access an element in a list, one has to iterate from a known position (like the beginning or the end) to that position.

```
#include <iostream> // cout
#include <list> // list
using namespace std; // namespace

list<int> L;
L.push_back(1);           // Insert a 1 integer at the end: [1]
L.push_front(2);          // Insert a 2 integer at the beginning: [2 1]
L.insert(++L.begin(),0); // Insert value 0 before position of first argument // [2 0 1]

L.push_back(5); // [2 0 1 5]
L.push_back(6); // [2 0 1 5 6]

list<int>::iterator i; // define iterator
for (i=L.begin(); i != L.end(); ++i) cout << *i << " ";
cout << endl;
```

list: doubly linked list
vector: elements contiguous
Not possible random access: list[]

C++ STL library: map

map container

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key.

In the example we use a key *string* that names the engineering branch (MEFT, MEEC,...) and a vector of data structures containing students data

C++ STL library: map (cont.)

```
#include <string>
#include <iostream>
#include <map>
#include <vector>
#include <utility>
using namespace std;

struct IST {
  string name; // nome
  float mark; // nota
  int id;
};

int main() {
  // define map element
  map<string, vector<IST> > M;
  M["MEFT"]; // empty vector created
  M["MEEC"];

  // fill vector structures
  IST A;
  A.name = "John Lob";
  A.mark = 15.5;
  A.id = 96000;
  // fill map
  M.find("MEFT")->second.push_back(A);
  // alternatively
  M["MEFT"].push_back(A);
  // fill another element
  A.name = "Tiago Num";
  A.mark = 17.0;
  A.id = 96001;
  M.find("MEFT")->second.push_back(A);

  // vector size
  cout << "MEFT vector size=''
       << M.find("MEFT")->second.size()
       << endl; // = 2

  //list map contents
  map< string,vector<IST> >::iterator it;
  for( it=M.begin(); it!=M.end(); ++it) {
     cout << it->first << ": "
          << it->second.size() << endl;
  }
  // retrieve vector MEFT
  vector<IST> meft=M.find("MEFT")->second;
  (...)
}
```


C++ STL library: stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

```
// stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main () {
  std::stack<int> mystack;
  for (int i=0; i<5; ++i) mystack.push(i);

  std::cout << "Popping out elements..." << std::flush;
  while (!mystack.empty())  {
     std::cout << ' ' << mystack.top(); //points to last element of stack
     mystack.pop(); //removes element on top of stack
  }
  std::cout << '\n';
}

Output:
Popping out elements... 4 3 2 1 0
```

C++11: range for loops

- ✔ C++11 augmented the for statement to support the paradigm of iterating over collections
- ✔ It makes the code much more simple and cleaner
- ✔ For observing the colection elements, use the following syntax:

```
for (const auto& elem : container)    // capture by const reference
for (auto elem : container)    // capture by value (to be used only on simple case
```

- ✔ For modifying the colection elements in place, use:

```
for (auto& elem : container)    // capture by (non-const) reference
for (auto&& elem : container)    // more general, capture by &&
```


C++11: range for loops examples

```
// loop on container values
vector<int> vec{10,20};
for (int i : vec ) {
  cout << i;
}
for (auto i : vec ) {
  cout << i;
}
```

```
// modifying contents of vector
vector<int> vec{10,20};
for (auto& i : vec ) {
  i++;
}

// loop over an array
int a[4]{1,2,3,4};
for(auto& i : a)
  std::cout << i << "\n";
```

```
// more complex: a map
map<string,int> m{{"a",10},{"b",20}};
for (auto e : m ) {
  cout << e.first << e.second;
}

// similar and more effective
// (no copying)
for (const auto& e : m ) {
  cout << m.first << " " << m.second;
}
cout << endl;
```

```
// over a brace-init-list
// (std::initializer_list)
for(auto i : {1,2,3,4}) {
  std::cout << i << " ";
}
cout << endl;
```