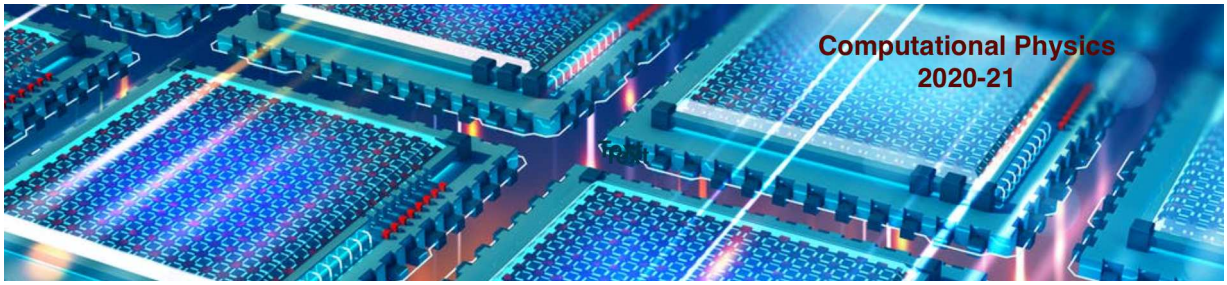
# Computational Physics

***numerical methods with C++ (and UNIX)***

***2020-21***




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



## C++ STL library: algorithms

- ✔ STL library includes data handling functions able to perform searching, removing, sorting and copying
- ✔ algorithms operate on containers via iterators
- ✔ algorithms are defined through header *<algorithm>*
- ✔ algorithms are usually divided in three categories:

  **mutating:** copying, exchanging, removing, reversing, rotating, filling

  **non-modifying:** looping on every container element, searching match

  **sorting, searching:** find minimum and maximum, copy sorting

# C++ STL library: algorithms

| non-modifying sequence operations | |
|---|---|
| **all_of** | Test condition on all elements in range |
| **any_of** | Test if any element in range fulfills condition |
| **none_of** | Test if no elements fulfill condition |
| **for_each** | Apply function to range |
| **find** | Find value in range |
| **find_if** | Find element in range |
| **find_if_not** | Find element in range (negative condition) |
| **find_end** | Find last subsequence in range |
| **find_first _of** | Find element from set in range |
| **adjacent_find** | Find equal adjacent elements in range |
| **count** | Count appearances of value in range |
| **count_if** | Return number of elements in range satisfying condition |
| **mismatch** | Return first position where two ranges differ |



# C++ STL library: algorithms

| non-modifying sequence operations (cont.) | |
|---|---|
| **equal** | Test whether the elements in two ranges are equal |
| **is_permutation** | Test whether range is permutation of another |
| **search** | Search range for subsequence |
| **search_n** | Search range for elements |

# C++ STL: non-modifying examples

## for_each

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

int main() {
  vector<int> v {0,1,2,3,4,5,6};

  // print all elements of vector
  for_each( v.begin(), v.end(), [ ](int  n){ std::cout<<n << " ";});
  cout << endl;

  // accumulate sum of all elements
  auto sum = 0.;
  for_each( v.begin(), v.end(), [&sum](int  n){sum +=n;});
  cout << "sum=" << sum << endl;

  (...)
```



# C++ STL: non-modifying examples

## for_each

```
  (...)

  // compute mean
  auto n = v.end()-v.begin();
  auto mean = sum/n;
  cout << "n=" << n << " mean=" << mean << endl;

  // standard deviation
  double var = 0.;
  for_each( v.begin(), v.end(), [&var, mean](int  n){
    var +=pow(n-mean,2.);
  });
  var /= n; // unbiased estimator uses n-1 instead of n
  cout << "var=" << var << " std=" << sqrt(var) << endl;
}
```

# C++ STL: non-modifying examples

## find

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find
#include <vector>       // std::vector

int main () {
  // using std::find with array and pointer:
  int x[] = { 10, 20, 30, 40 };

  // find returns pointer to first element found
  // otherwise, returns pointer to end of array
  int *p = std::find (x, x+4, 30);
  if (p != x+4) {
    std::cout << "Element found in array: " << *p
              << " elem index=" << p-x << '\n';
  } else {
    std::cout << "Element not found!!!\n";
  }
```



# C++ STL: non-modifying examples

## find

```
  (...)

  // using std::find with vector and iterator:
  std::vector<int> v(x,x+4);

  // find
  std::vector<int>::iterator it = find (v.begin(), v.end(), 40);
  if (it != v.end()) {
    std::cout << "Element found in vector: " << *it
              << " index=" << it-v.begin() << '\n';
  } else {
    std::cout << "Element not found in vector\n";
  }
  return 0;
}
```

# C++ STL library: algorithms

| modifying sequence operations | |
|---|---|
| **copy** | copy range of elements |
| **copy_n** | Copies the first n elements from the range beginning at first into the range beginning at result |
| **copy_if** | Copy certain elements of range upon function result |
| **copy_backward** | Copy range of elements backward |
| **move** | Move range of elements |
| **move_backward** | Move range of elements backward |
| **swap** | Exchange values of two objects |
| **swap_ranges** | Exchange values of two ranges |
| **iter_swap** | Exchange values of objects pointed to by two iterators |
| **transform** | Transform range |
| **replace** | Replace value in range |



# C++ STL library: algorithms

| modifying sequence operations (cont.) | |
|---|---|
| **replace_if** | |
| **replace_copy** | copy range replacing value |
| **fill** | Fill range with value |
| **generate** | Generate values for range with function |
| **remove** | Remove value from range |
| **remove_if** | |
| **unique** | Remove consecutive duplicates in range |
| **unique_copy** | Copy range removing duplicates |
| **reverse** | reverse range |
| **rotate** | Rotate left the elements in range |
| **random_shuffle** | Randomly rearrange elements in range |

# C++ STL: modifying examples

## transform

```cpp
/*
std::transform applies the given function to a range and stores the
result in another range
*/

#include <iostream>     // std::cout
#include <algorithm>    // std::transform
#include <vector>       // std::vector
#include <functional>   // std::plus (described later on slides)

int main () {
  std::vector<int> foo;
  std::vector<int> bar;

  // set some values:
  for (int i=1; i<6; i++)
  foo.push_back (i*10); // foo: 10 20 30 40 50

  (...)
```



# C++ STL: modifying examples

## transform (cont.)

```cpp
  (...)

  // allocate space dynamically to vector
  bar.resize(foo.size());

  // transform foo values by adding 1, and write to another vector
  std::transform (foo.begin(), foo.end(), bar.begin(), [](x){x++;});
  // bar: 11 21 31 41 51

  // std::plus adds together its two arguments:
  std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(),
                                                 std::plus<int>());
  // foo: 21 41 61 81 101

  std::cout << "foo contains:";
  for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
     std::cout << ' ' << *it;
  std::cout << '\n';
}
```

# C++ STL: modifying examples

## copy

```
#include <iostream>     // std::cout
#include <algorithm>    // std::copy
#include <vector>       // std::vector

int main () {
  int x[]={10,20,30,-40,50,-60,70};
  std::vector<int> v(7);

  // returns iterator to the end of the destination range
  // where elements have been copied
  auto a = std::copy ( x, x+7, v.begin() );

  std::cout << "vector contains:";
  for (std::vector<int>::iterator it = v.begin(); it!=v.end(); ++it)
     std::cout << ' ' << *it;
  std::cout << '\n';
```



# C++ STL: modifying examples

## copy (cont.)

```
  std::cout << "number of elements between two iterators: "
            << std::distance(v.begin(),a) << std::endl;
  for (auto index = a-v.begin()-1; index>=0; --index) {
    std::cout << index << " v=" << v[index] << " | " ;
  }
  std::cout << '\n';

  // new vector to house copy of first 3 elements of myints
  std::vector<int> y;
  y.resize(3); // allocate space to 3 elements
  std::copy_n ( x, 3, y.begin() );

  // copy conditional (only negative numbers)
  // copy only negative numbers (use unary predicate)
  std::vector<int> z(7); // prepare to max size
  auto it = std::copy_if( x, x+7, z.begin(),
                          [](int i){return (i<0);} );
  bar.resize(std::distance(z.begin(),it));  // shrink container
                                            // to new size
}
```

# C++ STL library: algorithms

| sorting | |
|---|---|
| **sort** | sort elements in range |
| **stable _sort** | Sort elements preserving order of equivalents |
| **partial _sort** | Partially sort elements in range |
| **partial _sort _copy** | Copy and partially sort range |
| **is _sorted** | Check whether range is sorted |
| **is _sorted _until** | Find first unsorted element in range |



# C++ STL library: algorithms

| min/max | |
|---|---|
| **min** | Return the smallest |
| **max** | Return the largest |
| **minmax** | Return smallest and largest elements |
| **min _element** | Return smallest element in range |
| **max _element** | Return largest element in range |
| **minmax _element** | Return smallest and largest elements in range |

# C++ STL: sorting-searching examples

## sorting array elements

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

int main() {
  vector<int> v = {56, 32, -43, 23, 12, 93, 132, -154};

  // descending order: uses binary predicate
  sort(v.begin(), v.end(), [](int a, int b){ return a>b; } );

  // sorted vector (it changed)
  for (int i : v) cout << i << " ";
  cout << endl;

  // ascending order (default behaviour without function)
  sort(v.begin(), v.end(), [](int a, int b){ return a<b; } );

  // sorted vector (it changed)
  for (int i : v) cout << i << " ";
  cout << endl;
}
```

```
132 93 56 32 23 12 -43 -154
-154 -43 12 23 32 56 93 132
```



# C++ STL: sorting-searching examples

## sorting array elements

```
  // sort a vector contents for a given range
  int myints[] = {32,71,12,45,26,80,53,33};
  vector<int> v(myints, myints+8); // 32 71 12 45 26 80 53 33
  sort(v.begin(), v.begin()+4); //(12 32 45 71)26 80 53 33

  // find maximal element
  // return iterator to element found
  float max = *( max_element( v.begin(), v.end() ) );
```

# C++ STL library: numeric

✔ C++ numeric algorithms exist in header **<numeric>**, defined on **std** namespace

| numeric functions | |
|---|---|
| **accumulate** | sum the elements |
| **adjacent_difference** | make each output element be the difference between input element and previous element |
| **inner_product** | compute inner product |
| **for_each** | Apply function to range |
| **partial_sum** | each output element will be the sum of the corresponding input element and all previous elements. |



# C++ STL library: numeric examples

**sum of array elements**

```
1 /* sum */
2 double u[3] = {1.1, 2.2, 3.3};
3 double v[3] = {11.1, 22.2, 33.3};
4 double sum = accumulate(u, u+3, 0.0); // init value=0.
5 printf("sum=%f \n", sum);
```

```
sum=6.600000
```

**array element calculations applying function**

```
1 // lambda function parameters: current sum, next element
2 vector<int> d={1, 2, 3};
3 int n = accumulate(d.begin(), d.end(), 0, [](int x1, int x2) {
4   printf("x1=%f x2=%f x1*10+x2=%f \n", x1, x2, x1*10+x2);
5   return x1 * 10 + x2;
6 });
```

```
a=0 d=1 a*10+d=1

a=1 d=2 a*10+d=12

a=12 d=3 a*10+d=123
```

# C++ STL library: functors

- The Standard Library contains a number of predefined function objects to be used with the STL algorithms
- To use the functors, you must include the header *<functional>*

**arithmetic functors**

| Function Object | Operation on class objects |
|---|---|
| plus<class name> | *arg1 + arg2* |
| minus<class name> | *arg1 - arg2* |
| multiplies<class name> | *arg1 * arg2* |
| divides<class name> | *arg1 / arg2* |
| modulus<class name> | *arg1 % arg2* |
| negate<class name> | *-arg1* |



# C++ STL library: functors (cont.)

```
1  #include <iostream>     // std::cout
2  #include <functional>   // std::plus, ...
3  #include <algorithm>    // std::transform
4
5  int main() {
6   vector<int> a(100,1); // 100 numbers = 1
7   vector<int> b(a); // copy of a to b
8   vector<int> c(a.size()); // result vector
9
10  // add every two elements of vectors a and b and put result on c
11  transform(a.begin(), a.end(), b.begin(), c.begin(), plus<int>());
12
13  // subtract every two elements of vectors a and b and put result on c
14  transform(a.begin(), a.end(), b.begin(), c.begin(), minus<int>());
15
16  // negate elements of vector (50 1st elements)
17  transform(a.begin(), a.begin()+50, a.begin(), negate<int>());
18 }
```

# C++ STL library: functors (cont.)

➜ A **predicate** is a function object that **returns a Boolean**, or a value that can be converted to a Boolean

**comparison functors**

| Function Object | Operation on class objects |
|---|---|
| equal_to<class name> | *arg1 == arg2* |
| not_equal_to<class name> | *arg1 != arg2* |
| greater<class name> | *arg1 > arg2* |
| less<class name> | *arg1 < arg2* |
| greater_equal<class name> | *arg1 >= arg2* |
| less_equal<class name> | *arg1 <= arg2* |



# C++ STL library: functors (cont.)

```
1  // define array
2  int data[] = { 800, 250, 250, 100, 500, 500, 400 };
3
4  // create and initialize vectors to hold the debts
5  vector<int> v( data, data + sizeof( data ) / sizeof( data[0] ));
6
7  // sort into descending order
8  sort( v.begin(), v.end(), greater<int>() );
```

# C++ STL library: functors (cont.)

- **Binders** take a function object of two parameters and convert it to a functor that just accepts one argument
- One of the two arguments is *frozen*, i.e., is always passing the same value to that argument

## Binders

| Function Object | Operation on class objects |
|---|---|
| bind1st | hold first argument of a functor constant |
| bind2nd | hold second argument of a functor constant |



# C++ STL library: functors (cont.)

```
1    //count number of elements = 10
2   int x[] = {10,20,30,10,40,50,10};
3   int cx = count_if (x, x+7, bind1st(equal_to<int>(),10) );
4   cout << "Number of 10 elements = " << cx << " \n";
5
6   // count number of elements greater than 30
7   cx = count_if (x, x+7, bind2nd(greater<int>(),30) );
8   cout << "Number of elements (>30) = " << cx << " \n";
9  }
```

# C++ const declaration

- The *const* declaration allows to avoid further changes on variables or pointers
- *const* variables shall be initialized when declared
- constant value

```
int const MyVariable = 0; //const applies to the left declaration (int)
const int MyVariable = 0; //do the same (nothing on left => right declaration)
MyVariable = 10; //compiler error, value cannot be changed
```

```
int const * pMyVar1 = NULL; // ERROR, because not initialized

int i = 10;                 // GOOD
int const * pMyVar1 = &i;
```

- constant pointer

```
int i=10, j=10;
int* const pMyVar2 = &i; //const pointer to variable i
pMyVar2 = &j; // can it be done????? (ERROR)
```



# C++ const correctness (cont.)

- **constant pointer to constant value**

```
int i = 10;
int const * const q = &i;
```

It will not be possible to change the address and the value pointed to!

- **constant functions**

concept applied to class member functions

any const member function that attempts to *change a member variable* or call a non-const member function will cause a compiler error to occur

*const class objects* can only explicitly call *const member functions*

```
class T {
  public:
    ...
    void bar() const;
  private:
  int i;
};
```

```
// ********* Implementation
void T::bar() const {
  i=100; //ERROR, the object cannot be changed!
 }
```

# C++ const correctness (cont.)

- constant references

  we want to pass an object as argument of a function in an optimized (light) way => by reference

  we want to avoid any modifications of my object!

```
class T {
  public:
    ...
    void bar(const T&) const;
  private:
  int i;
};
```