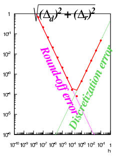
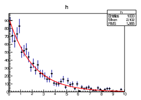
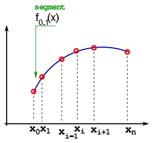
Computational Physics

numerical methods with C++ (and UNIX)

2018-19







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

C++ Input / Output

- ✔ The *iostream* library allow us to enter data from keyboard and display data on monitor

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

  ...
  // read several  real values from the keyboard
  float a, b, ...;
  cin >> a >> b >> ...;

  // read a string from keyboard (no blank spaces)
  string s;
  cin >> s;

  // read a full line (including blank spaces)
  string s;
  getline(cin, s);

  // output line
  cout << s << endl;
  cout << s << "\n"; //similar to previous line
```

C++ Input / Output (cont.)

- ✔ The *fstream* library allow us read from and write to files

```cpp
// read from file

#include <fstream>
using namespace std;

  ...
  // declare input file stream and open "filename.dat" file
  ifstream F;
  F.open("filename.dat");  //shortly could be: ifstream F("filename.dat");

  // read file values
  int i=0;
  double a[10];
  while (F>>a[i] && i<10) { // logical true if reading OK
    cout << i << " " << a[i] << endl;
    i++;
  }

  F.close(); // close file
```

C++ Input / Output (cont.)

- ✔ The *fstream* library allow us read from and write to files

```cpp
// write to file

#include <fstream>
using namespace std;

  ...
  // declare output file stream and open "filename.dat" file
  ofstream F("filename.dat");

  // output values read before to file
  int i=0;
  double a[10];
  while (i<10) { // logical true if reading OK
    cout << i << " " << a[i] << endl;
    F << a[i];
    i++;
  }

  F.close(); // close file
```

C++ Input / Output (cont.)

✔ The *fstream* library allow us read from and write to files

```cpp
// read and write to file

#include <fstream>
using namespace std;

  ...
  // declare output file stream and open "filename.dat" file
  fstream F("filename.dat", ios::in | ios::out | ios::app); //app=if file exists write at end

  // output values read before to file
  int i=0;
  double a[10];
  while (i<10) { // logical true if reading OK
    cout << i << " " << a[i] << endl;
    F << a[i];
    i++;
  }

  F.close(); // close file
```

C++ output formatting

✔ Formatted output can be done using the C-style *cstdio* library

```cpp
  printf("formatted output: integer=%d float=%f float=%12.3f\n",a,b,c); // try \n\a
```

✔ The input/ouput *iomanip* library allow us to print data in formatted way

✔ The width of the decimal part (including the decimal point) is given by *setprecision(n)* and total width is given by *setw(n)*

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
  ...
  double pi = 3.14159265358;
  cout << setprecison(7) << setw(10) << pi << endl;
```

The number 3.141592 would be printed!

C++ output formatting (cont.)

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <cmath> // M_PI
#include <cstdio>
int main() {
  printf(``1) %28.26f\n'',M_PI);
  cout << ``2) `` << M_PI << endl;
  cout << ``3) `` << setprecision(27) << M_PI << endl;
  cout << ``4) `` << setiosflags(ios::scientific) << M_PI << endl;
  cout << setiosflags(ios::scientific) << setprecision(5);
  cout << ``5) `` << M_PI << endl;

  cout << resetiosflags(ios::scientific);
  cout << setprecision(15) << setiosflags(ios::fixed | ios::showpoint) <<
       endl;
  for (int i=0; i<4; i++) {
    cout << i << `` `` << sin(M_PI/(double)((i+1))) << endl;
  }
}
```

```
1) 3.14159265358979311599796347
2) 3.14159
3) 3.14159265358979311599796347
4) 3.141592653589793115997963469e+00
5) 3.14159e+00
```

```
0 0.000000000000000
1 1.000000000000000
2 0.866025403784439
3 0.707106781186547
```

C++ dynamic memory allocation

- ✔ In a C++ program memory can be allocated dynamically at running time through the *new* operator and is responsability of the user to delete it through the *delete* operator (otherwise remain there through all the program execution!)
- ✔ Memory is allocated by using the *new* operator followed by a data type and it returns a pointer to the first elemnt of the sequence

```
float *f = new float; // memory allocated for 1 float
*f = 2.354; // value set

float *fv = new float[10]; // memory allocated for 10 floats
fv[0] = 2.345; //1st element set
*(fv+1) = 3.245; // 2nd element
```

- ✔ To free memory the operator *delete* is used folowed by the pointer to the object

```
delete f; //memory is freed (or deallocated)

delete[] fv; // the destructors are called for every object
```

- ✔ To obtain in linux, information about memory occupation in MBytes
  > free -m

C++ dynamic memory alloc: exception

- ✔ An exception of type *bad_alloc* is thrown when the memory allocation fails
- ✔ The simplest way of controlling if the memory was properly allocated is to avoid the *Exception* to occur and check if a null pointer is returned

```
#include <cstdlib> //exit()
#include <new> //std::nothrow
...
// allocated memory for 10 floats
float *fv = new (nothrow) float[10];
if (fv != NULL) { // check for null pointer
  fv[0] = 2.345; //1st element set
  *(fv+1) = 3.245; // 2nd element
  *(fv+2) = 2.46; // 3rd element
  ...
} else {
  exit(1);
}
```

C++ dynamic memory alloc examples

An array of 10 objects is allocated
The *delete []* operator will call the destructors of every object of the array

```
class A {
  public:
  A() {printf("%s ",
         __PRETTY_FUNCTION__);}
 ~A() {printf("%s ",
         __PRETTY_FUNCTION__);}
};

int main() {
 // create array of objects
 A *a = new A[10];
 // deallocate
 // object destructor is called
 delete [] a;
}
```

An array of 10 pointers to objects is allocated
The *delete []* operator **will not call** the destructors of every object of the array; do not forget what we store were pointers!

```
int main() {
// create array of pointers objects
A **a = new A*[10];
// create objects
for (int i=0; i<10; i++) {
 a[i] = new A();
}
// deallocate
// call destructor
for (int i=0; i<10; i++) {
 delete a[i];
}
delete [] a;
}
```

C++ STL library

- ✔ **Containers**
  A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, 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 adaptors. Container adaptors 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

- ✔ **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
```

string class

- ✔ Strings are objects that represent sequences of characters.
- ✔ The standard string class provides support for such objects with an interface similar to that of a standard container of bytes, but adding features specifically designed to operate with strings of single-byte characters.

```
  #include <iostream>
  #include <string>
  int main (){
    std::string str="We think in generalities, but we live in details.";
    std::string str2 = str.substr (3,5);  //"think"
    std::size_t pos = str.find(‘‘live’’); // position of "live" in str
    std::string str3 = str.substr(pos);   // get from "live" to the end
    std::cout << str2 << ’ ’ << str3 << ’\n’;
    return 0;
  }
```

C++ STL library (cont.)

- 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*

- **vector container**
  similar to an array but can be dinamically enlarged or shrinked

```
#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);
    }
    cout << "vector size=" << vec.size() << endl; // vector size
```

C++ STL library (cont.)

- **vector container (cont.)**

```
    // 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++;
    }

    // sort a vector contents and another way of inserting vector values
    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
    float max = *( max_element( v.begin(), v.end() ) ); //iterator

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

C++ vector (cont.)

```
// an empty vector of integers
   vector <int> v;
   vector<int> v1(5); // a vector with 5 elements, each an integer

// 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 the size and values of v2
   vector<int> v3(v2);

// A vector with the size and values of v2
   vector<int> v4(v2.begin(),v2.end());

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

C++ vector (cont.)

```
// 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 vectors
  typedef vector<vector<int> > vecM;
  vecM matrix;

// Create a 2 x 5 matrix
// ...First, create a vector row vector (5 elem)
  vector<int> vr(5, 15);
// ...Now create a vector of 2 elements with each element a copy of v2
  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 evenry row elem
        cout << vm[i][j] << " ";}
    cout << endl;
  }

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