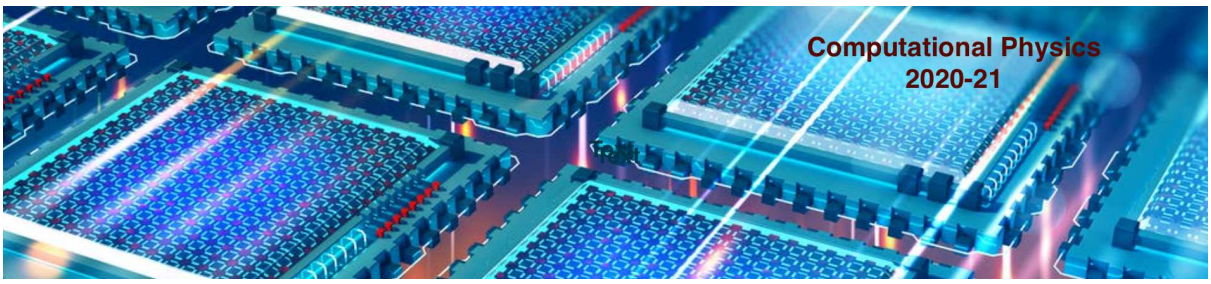
Computational Physics

numerical methods with C++ (and UNIX)

2020-21



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

Computational Physics

C++

An object oriented language

Fernando Barao, Phys Department IST (Lisbon)

C++ functions arguments

✔ Passing by value

A copy of the variable is made and passed to the function. Any modification of the variable inside the function will be local and lost at return!

```
1  double sum(double, double); // function prototyping
2
3  int main() {
4     // initialize variables
5     double a=0., b=5.43;   // passing variables to sum by copy
6     double a = sum(a, b); // result was also returned by copy
7     return 0;
8  }
9
10 double sum(double a1, double a2) {
11    return a1+a2; //copy of the result returned
12 }
```


C++ functions arguments (cont.)

✔ Passing by pointer

The memory address of the variable is passed to the function and therefore the variable contents inside the function can be modified.

```
1  // - The result of the factorial() function is passed to the main program
2  // through a pointer to a double;
3  // - The double variable is initialized in the main program
4
5  void factorial(int, double*); // pointer to double is passed
6  int main() {
7    int n=10; double d = 1.;
8    factorial(n, &d);
9    return 0;
10 }
11 void factorial(int n, double* pd) {
12   double fact = *pd;
13   for (int count=n; count > 0; --count) {
14     fact *= (double)count;
15   }
16 }
```

C++ functions arguments (cont.)

✔ Passing by reference

Similar to the pointer passing but more symbolic!

```
1  // - The result of the factorial() function is passed to the main
2  // program through a reference to the address of a variable (pointer);
3  // - The double variable is initialized in the main program
4
5  void factorial(int, double&); // double address reference is passed
6  int main() {
7    int n=10; double d = 1.;
8    factorial(n, d);
9    return 0;
10 }
11 void factorial(int n, double& fact) {
12   for (int count=n; count > 0; --count) {
13     fact *= (double)count;
14   }
15 }
```


C++ functions arguments (cont.)

✔ Passing arrays

Unlike scalar variables, arrays cannot be passed by value. Pointer has to be used.

```
1  // - A one dimensional array containing values of integers
2  // is passed to function factorial()
3  // - Result is also returned on an array passed by pointer
4
5  void factorial(int, int*, double*); // passing pointer
6  int main() {
7    int vi[4]={10,8,4,10}; // dim-4 array initialized
8    double vr[4] = {0.};
9    factorial(4,vi,vr);
10   return 0;
11 }
12 void factorial(int n, int* vi, double* vr) {
13   for (int i=0; i<n; i++) {
14     for (int count=vi[i]; count > 0; --count) {
15       vr[i] *= (double)count;
16     }
17   }
18 } //MISTAKE on the FACTORIAL CALCULATION???
```

C++ functions arguments (cont.)

✔ **default argument value**

In the prototyping of the function a default value to arguments can be defined.

```
1  // A one dimensional array containing values of integers
2  // is passed to function factorial()
3
4  void factorial(int *p=NULL, double *pd=NULL, int n=4);
5  int main() {
6    int vi[4]={10,12,15,22}; // dim-4 array initialized
7    double vr[4] = {0.};
8
9    factorial(); // by default, the value n=4 will be passed
10                //and the NULL pointers
11
12   factorial(vi,vr); // the value n=4 will be passed by default
13                     //and the valid pointers
14
15   return 0;
16 }
```


C++ function overloading and recursive calling

✔ **function overloading:** Excepting the *main()* function, two entirely different functions are allowed to have the same name, provided they have **distinct list of arguments**

```
1  // two same name functions prototyping
2  double factorial(int);
3  void factorial(int*, double*, int);
4
5  int main() {
6    int n=10; // variable to be passed
7    double a = factorial(n);
8
9    int vi[2] = {5, 7};
10   double vr[2] = {}; // init to zeros
11   factorial(vi, vr, 2);
12
13   return 0;
14 }
```

✔ **recursive calling:** C++ functions are allowed to call themselves

C++ program arguments

```
1  /*
2  The main() function may optionally have arguments which allow parameters to be
3  passed to the program from the operating system
4   */
5
6  #include <cstdio> //printf
7  #include <cstdlib> //atoi, atof
8
9  int main(int argc, char *argv[]) {
10
11   //retrieving character arrays
12   for (int i=0; i<argc; i++) { //argc= number of arguments + 1 (program name)
13     printf("argument number %d, %s\n", i, argv[i]);
14   }
15
16   //retrieving argument numbers
17   for (int i=1; i<argc; i++) { //argc= number of arguments + 1 (program name)
18     double a = atof(argv[i]);
19     printf("argument number %d, %10.2f\n", i, a);
20   }
21
22   return 0;
23 }
```


C++11: type inference and deduction

- ✔ C++11 introduces type inference capability using the **auto** keyword, which means that the compiler infers the type of a variable at the point of declaration
- ✔ It's very practical to use when we have a very long declaration to type, as it happens with STL container iterators
- ✔ Can also be used on function returns
- ✔ Don't forget to use the compiler option *-std=c++11*

```
1 auto a = 10; // integer
2 auto b = 10.; //double
3 string s("Teste à string");
4 auto si = s.begin(); // iterator
```

```
1 // iterating over container
2 // std::string::iterator -> auto
3 string s("Teste à string");
4 for ( auto it=s.begin(); it!= s.end(); ++it ) {
5   std::cout << *it;
6 }
7 std::cout << endl;
```

```
1 // functions return
2 int& func();
3
4 auto x = func(); // x is a value
5 auto& x = func(); // x is a reference
6 const auto& x = func(); // x frozen
```

C++11: lambda functions

- ✔ lambda functions were introduced in C++11 revision
- ✔ a way of creating quickly-and-easyly functions (eventually to pass in to another function)
- ✔ How to create a lambda function?

```
[capture-list](params)->ReturnType {
// code
};

Lambda capture: You can capture variables by ref or value

[=] // capture all current available variables by value

[&] // capture all current available variables by ref

[var1, &var2] // capture var1 by value and var2 by ref
```


C++11: lambda functions examples

```cpp
#include <cstdio>

// define lambda function f1: no parameters, no return

auto f1 = []() {
  printf("f1 function: test lambda function \n");
};
// call function
f1();
```

```cpp
#include <string>
#include <iostream>
// - define lambda function that searchs for a pattern
//   in variable name that is defined outside function
// - we need to capture variables outside function (&, =)

std::string name="teste name";
auto f2 = [&name](std::string patt) {
   // returning boolean
   return name.find(patt) != std::string::npos;
};
auto a = f2("teste");
std::cout << a << std::endl;
std::cout << f2("xx") << std::endl;
```

C++ preprocessor directives

✔ A statement following the # character in a C++ code is a preprocessor directive

#include $< file >$	includes file at this location of the code
#define VAR 100	the preprocessor will replace the variable VAR by 100
#undef VAR	undefine VAR
#define getmax(a,b) $a > b?a : b$	the preprocessor will replace the symbolic code getmax() by the logical condition
#ifdef VAR ... #endif	conditional inclusions depending if VAR is defined
#ifndef VAR ... #endif	conditional inclusions depending if VAR is not defined
#if ... #elif ... #else ... #endif	conditional inclusions

Structuring your C++ code

When writing a program you can divide it into three parts:

✔ a *header file* containing the structure declarations and prototypes for functions that can be used by those structures
- ➜ function prototypes
- ➜ symbolic constants defined using #define or const
- ➜ structure declarations
- ➜ class declarations
- ➜ inline functions

✔ a *source code* file that contains the code for the structure-related functions

✔ a *main program*

C++ header files (cont.)

- ✔ A set of prototyping functions are already defined in header files **.h* and can be included through the preprocessor directive *#include <header file>*
- ✔ The *#include* statement asks the preprocessor to attach at the location of the statement a copy of the header file
- ✔ The C++ preprocessor runs as part of the compilation process

files	obs
iostream, cstdio, fstream, iomanip, iostream, strstream	**input/output**
cmath, complex, cstdlib, numeric, valarray	**mathematical**
string, cstring, cstdlib	**strings**
algorithms	**STL algorithms**
vector, list, map, queue, set, stack	**STL containers**
iterators	**STL iterators**
ctime, functional, memory, utility	**general**
cfloat, climits, csignal, ctime, cstdlib, exception	**language**