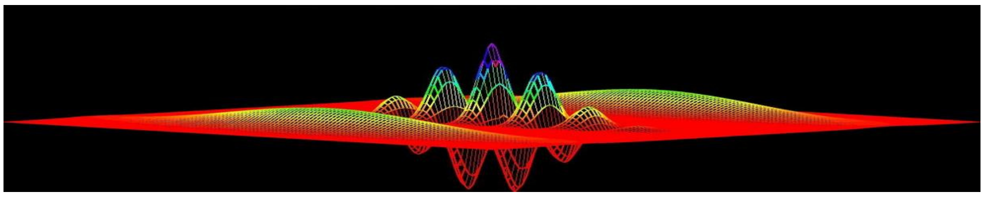
Computational Physics

numerical methods with C++ (and UNIX)



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: barao@lip.pt

Computational Physics

C++

An object oriented language

Fernando Barao, Phys Department IST (Lisbon)

C++ functions

- ✔ A function is a self contained program segment that carries out some specific, well defined task.
- ✔ Every C++ program consists of several functions, one of them mandatory : *main()*
- ✔ A function can return a value, values (arrays) or nothing.
- ✔ A function needs to be declared before being used ; *function prototyping* is needed if function come after

```

#include <cstdlib> // exit()
#include <cstdio> // printf

//function prototyping
double factorial(int);

/////////////////////////////////////////////
int main() {
  for (int i=0; i<=20; i++) {
    printf("factorial(%d)=%12.3e\n",i,factorial(i));
  }
  return 0;
}

/////////////////////////////////////////////
double factorial(int n) {
  double fact=1.;
  if (n<0) {exit(1);} //abort prog if n negative
  for (int count=n; count > 0; --count)
     fact *= (double)count;
  return fact;
}
```

C++ inline functions

Functions are used in C programs to avoid repeating the same block of code in many different places in the source.
Modular code is also easier to read and maintain.
However there is a price to pay : when a regular function is called in an executable, the program jumps to the block of memory in which the compiled function is stored, and then jumps back to its original position in memory when function returns.
Large jumps in memory space takes CPU time (operations of information saving and restoring are made at each call).
The *inline* keyword causes the piece of code composing the function to be placed where it is called

Inline functions are only useful for small functions (up to ~ 3 executable lines)

inline functions shall be defined in header files, because the compiler needs to access the function code before it is used

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 !

```
// The variable n is passed by value to the factorial()
    function

double factorial(int); // function prototyping
int main() {
  int n=10; // variable to be passed
  double a = factorial(n);
  return 0;
}
```

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.

```
// The result of the factorial() function is passed to the
// main program through a pointer to a double;
// the double is initialized in the main program.

void factorial(int, double*); // pointer to double is passed
int main() {
  int n=10; double d = 1.;
  factorial(n, &d);
  return 0;
}
void factorial(int n, double* pd) {
  double fact = *pd;
  for (int count=n; count > 0; --count) fact *= (double)count;
  //YOU SHOULD TRY IT!!! MISTAKE?
}
```

C++ functions arguments (cont.)

✔ **Passing by reference**

Similar to the pointer passing but more symbolic !

```
// The result of the factorial() function is passed to the main
// program through a reference to the address of a variable (
    pointer);
// the double is initialized in the main program.

void factorial(int, double&); // double address reference is
    passed
int main() {
  int n=10; double d = 1.;
  factorial(n, d);
  return 0;
}
void factorial(int n, double& fact) {
  for (int count=n; count > 0; --count) fact *= (double)count;
}
```

C++ functions arguments (cont.)

✔ **Passing arrays**

Unlike scalar variables, arrays cannot be passed by value. Reference or pointer have to be used.

```
// A one dimensional array containing values of integers
// is passed to function factorial()

void factorial(int, int*, double*); // passing pointer
int main() {
  int vi[4]={10,12,15,22}; // dim-4 array initialized
  double vr[4] = {0.};
  factorial(4,vi,vr);
  return 0;
}
void factorial(int n, int* vi, double* vr) {
  for (int i=0; i<n; i++) {
      for (int count=vi[i]; count > 0; --count) vr[i] *= (double)
          count;
  }
}
```

C++ functions arguments (cont.)

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

```
// A one dimensional array containing values of integers
// is passed to function factorial()

void factorial(int *p=NULL, double *pd=NULL, int n=4);
int main() {
  int vi[4]={10,12,15,22}; // dim-4 array initialized
  double vr[4] = {0.};

  factorial(); // by default, the value n=4 will be passed
               //and the NULL pointers

  factorial(vi,vr); // the value n=4 will be passed by default
                    //and the valid pointers

  return 0;
}
```

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

```
// two same name functions prototyping
double factorial(int);
void factorial(double*, double*, int);

int main() {
  int n=10; // variable to be passed
  double a = factorial(n);

  int vi[2] = {5, 7};
  double vr[2] = {}; // init to zeros
  factorial(vi, vr, 2);

  return 0;
}
```

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

C++ preprocessor directives

✔ A statement following the # character in a C++ code is a compiler of 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

C++ header files

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**

C++ program arguments

```
/*
The main() function may optionally have arguments which allow parameters to be
passed to the program from the operating system
*/

#include <cstdio> //printf
#include <cstdlib> //atoi, atof

int main(int argc, char *argv[]) {

  //retrieving character arrays
  for (int i=0; i<argc; i++) { //argc= number of arguments + 1 (program name)
    printf("argument number %d, %s\n", i, argv[i]);
  }

  //retrieving argument numbers
  for (int i=1; i<argc; i++) { //argc= number of arguments + 1 (program name)
    double a = atof(argv[i]);
    printf("argument number %d, %10.2f\n", i, a);
  }

  return 0;
}
```

C++ timing

- ✔ The header file *time.h* defines a number of library functions which can be used to assess how much CPU time a C++ program consumes during execution
- ✔ A call to the function *clock()* will return the amount of CPU time used so far
- ✔ To normalize the time to seconds the returned number shall be divided by the variable *CLOCKS_PER_SEC*, defined inside *time.h*
- ✔ Next example computes time per operation in microseconds spent in calculating x^4, in a direct way and through the *pow()* function

C++ timing (cont.)

```
#include <ctime> // clock()
#include <cmath> // pow()
#include <iostream> // cout
using namespace std;
#define N 1000000

int main() {
  double a=12345678967598.0, b; //variable declaration

  //compute time spent on power to the fourth the double
  clock_t time1 = clock();
  for (int i=0; i<N; i++) b=a*a*a*a;
  clock_t time2 = clock();
  double dtime1 = (double)(time2-time1)/(double)CLOCKS_PER_SEC;

  //...using pow
  clock_t time1 = clock();
  for (int i=0; i<N; i++) b=pow(a,4.);
  clock_t time2 = clock();
  double dtime2 = (double)(time2-time1)/(double)CLOCKS_PER_SEC;

  cout << dtime1 << `` | `` << dtime2 << endl;
  return 0;
}
```

C++ random numbers

- ✔ Some calculations require the use of random numbers like the Monte-Carlo calculations
- ✔ The system header file *stdlib.h* provides the function *rand()* that returns a random integer (fairly good approximation) in the range *[0, RAND_MAX]*
- ✔ The sequence seed can be fixed through a call to *srand(int)* rendering therefore the random sequences repeatable
  by default, rand() is seeded with the value 1
- ✔ To generate independent sequences a common practice is to use the current UNIX time (number of seconds elapsed since January 1st, 1970)
  time(NULL) returns an integer
- ✔ The next example produces a sequence of 10^5 values between 0 and 1

C++ random numbers (cont.)

```cpp
#include <ctime> // time()
#include <cstdlib> // rand()
#include <iostream> // cout
using namespace std;

int main() {
  //set random seed
  srand(time(NULL));

  //generate random values and compute mean and variance
  double sum=0.;
  double var=0.;
  for (int i=0; i<100000; i++) {
    double x = (double)rand()/(double)RAND_MAX;
    sum += x;
    var += (x-0.5)*(x-0.5);
  }
  double mean = x/100000.;
  var /= 100000.;

  cout << mean << `` | `` << var << "(expected variance = 1/12) WHY???'' << endl;
  return 0;
}
```

C++ complex numbers

- ✔ complex numbers are implemented in C++ through the complex class

```
#include <complex> //C++ standard library
using namespace std;

int main() {
  complex<double> Z(2.5, 4.0);
  double Zmod = abs(Z);
  double Zr = z.real();
  double Zi = z.imag();
  complex<double> Zc = conj(Z);
}
```

- ✔ C++ example of using complex class : Tcomplex.C