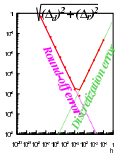
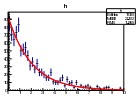
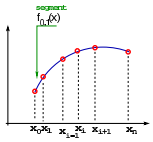
Computational Physics

numerical methods with C++ (and UNIX)

2018-19



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

C++ variables

✔ **Global variables**

They are defined outside the main function and user defined functions.
They are available to the program and user functions.

```
1 int n; // global variable
2 double factorial(); //function prototyping
3 int main() {
4  for (n=0; n<=20; n++)
5    {printf("factorial=%12.3e\n",n,factorial());}
6  return 0;
7 }
```

✔ **Local variables**

Variables defined inside the functions and private to them or within C++ code blocks { ... }.

The return from the function frees the local variable locations (lost)!

C++ variables (cont.)

✔ **Static variables**

Variables defined inside the functions can be declared as *static* and therefore their value is preserved between calls to the function. Mechanism that can be used to run code only once.

```
1 double F(int n) { //function code
2   static int initflag = 0;
3   if (!initflag) {
4     do initialization statements;
5     initflag++;
6   } // just run once
7 }
```


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 // The variable n is passed by value to the factorial()
      function
2
3 double factorial(int); // function prototyping
4 int main() {
5   int n=10; // variable to be passed
6   double a = factorial(n);
7   return 0;
8 }
```

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
2  // main program through a pointer to a double;
3  // the double 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) fact *= (double)count;
14   //YOU SHOULD TRY IT !!! MISTAKE?
15 }
```


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 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) fact *= (double)count;
13 }
```

C++ functions arguments (cont.)

✔ Passing arrays

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

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

```
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++ 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.)

```
1  #include <ctime> // clock()
2  #include <cmath> // pow()
3  #include <iostream> // cout
4  using namespace std;
5  #define N 1000000
6
7  int main() {
8    double a=12345678967598.0, b; //variable declaration
9
10   //compute time spent on power to the fourth the double
11   clock_t time1 = clock();
12   for (int i=0; i<N; i++) b=a*a*a*a;
13   clock_t time2 = clock();
14   double dtime1 = (double)(time2-time1)/(double)CLOCKS_PER_SEC;
15
16   //...using pow
17   clock_t time1 = clock();
18   for (int i=0; i<N; i++) b=pow(a,4.);
19   clock_t time2 = clock();
20   double dtime2 = (double)(time2-time1)/(double)CLOCKS_PER_SEC;
21
22   cout << dtime1 << `` | `` << dtime2 << endl;
23   return 0;
24 }
```


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.)

```
1  #include <ctime> // time()
2  #include <cstdlib> // rand()
3  #include <iostream> // cout
4  using namespace std;
5
6  int main() {
7    //set random seed
8    srand(time(NULL));
9
10   //generate random values and compute mean and variance
11   double sum=0.;
12   double var=0.;
13   for (int i=0; i<100000; i++) {
14     double x = (double)rand()/(double)RAND_MAX;
15     sum += x;
16     var += (x-0.5)*(x-0.5);
17   }
18   double mean = x/100000.;
19   var /= 100000.;
20
21   cout << mean << `` | `` << var << "(expected variance = 1/12) WHY???'' << endl;
22   return 0;
23 }
```


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

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

```
// 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

```
1  // write to file
2
3  #include <fstream>
4  using namespace std;
5
6    ...
7    // declare output file stream and open "filename.dat" file
8    ofstream F("filename.dat");
9
10   // output values read before to file
11   int i=0;
12   double a[10];
13   while (i<10) { // logical true if reading OK
14     cout << i << " " << a[i] << endl;
15     F << a[i];
16     i++;
17   }
18
19   F.close(); // close file
```


C++ Input / Output (cont.)

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

```
1  // read and write to file
2
3  #include <fstream>
4  using namespace std;
5
6    ...
7    // declare output file stream and open "filename.dat" file
8    fstream F("filename.dat", ios::in | ios::out | ios::app); //app=if file
         exists write at end
9
10   // output values read before to file
11   int i=0;
12   double a[10];
13   while (i<10) { // logical true if reading OK
14     cout << i << " " << a[i] << endl;
15     F << a[i];
16     i++;
17   }
18
19   F.close(); // close file
```