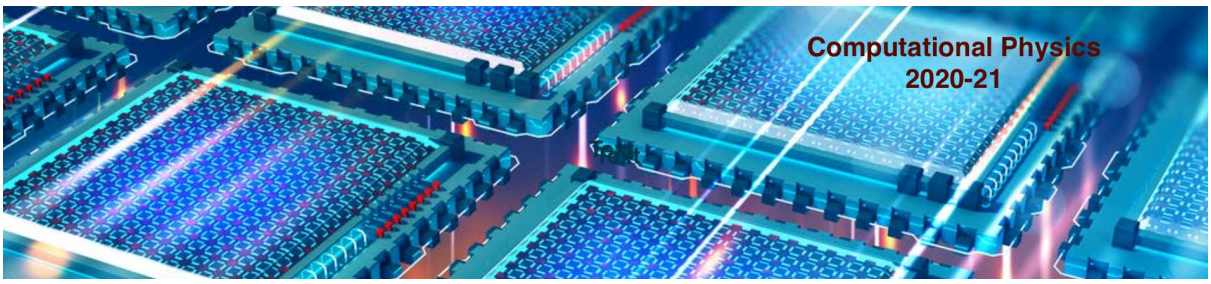
# Computational Physics

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

## *2020-21*




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



# Computational Physics

# C++

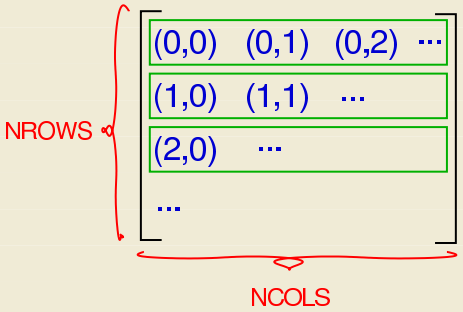
## An object oriented language

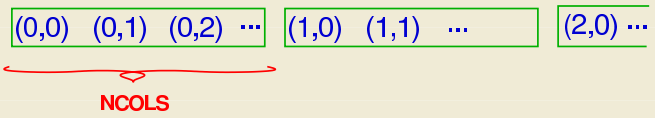
Fernando Barao, Phys Department IST (Lisbon)

# C++ arrays storage

✔ multi-dimensional arrays creation:
- as *rectangular sequential arrays* where rows are sequentially stored in memory
- as *arrays of arrays* (to be seen later on!)

```
1   // 1-dim array
2   double v[30] = {0}; //init to zero all
3   cout << "1st element='' << v[0] << endl;
4
5   // matrices
6   // 2-dim array declaration and init
7   // m[ROWS][COLS] : 10 ROWS * 5 COLS elements
8   int m[10][5] = {
9     {0, 1, 2, 3, 4}, //row 0
10    {5, 6, 7, 8, 9}, //row 1
11    {10, 11, 12, 13, 14}, //row 2
12    ...
13    {45, 46, 47, 48, 49} //last row
14  };
15
16  // print elements in memory order (see pointers section)
17  for (int i=0; i<50; i++) {
18    printf("%d ",*m);
19    m++;
20  }
```

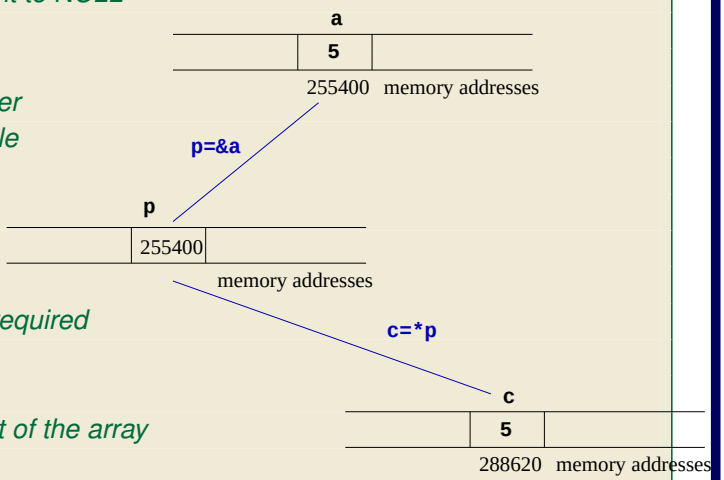





# C++ pointers

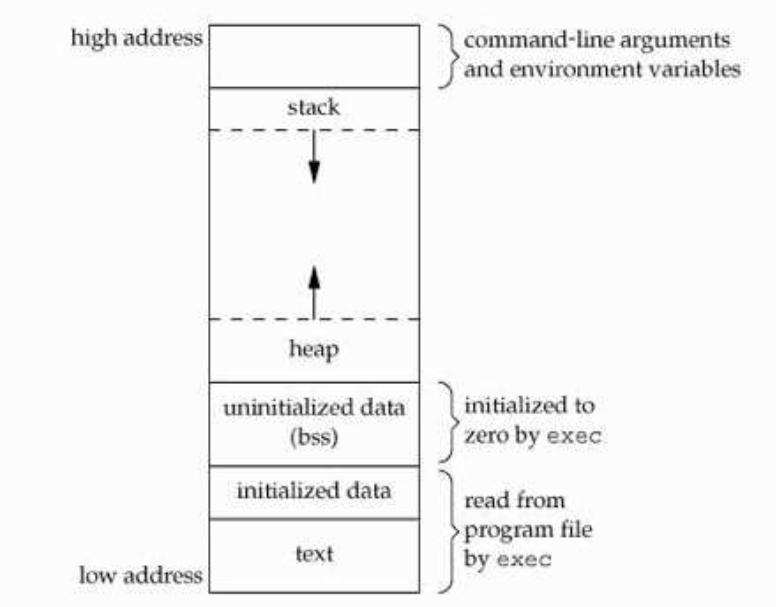
```
1   // declare pointer to an integer variable and set it to NULL
2   int *p = NULL; // nullptr from C++11 on...
3
4   // assign address of an integer number to pointer
5   int a = 5; p = &a; // p points to a variable
6
7   // deassign pointer: get value pointed to
8   int c = *p; // c=5
9   printf("%p \n", p); // print address
10                      // include <cstdio> required
11
12  // arrays and pointers:
13  // the array name is a pointer to the 1st element of the array
14  float v[10];
15  float a = *v; //retrieves 1st element of the array (float a=v[0];)
16  float *p2 = &v[1]; //pointer to 2nd element (similar to float *p2 = v+1);
17
18  // passing array to a function by reference/pointer
19  float a[100];
20  function(a); //function prototype: void function(float [])
21
22  // character string pointer
23  char *word = "four"; //similar to: char word[5]="four"
```



# C++: program memory

The memory of a programm is typically divided into different areas, called segments:

- ✔ The *code segment* (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- ✔ The *bss segment* (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- ✔ The *data segment* (also called the initialized data segment), where initialized global and static variables are stored.




- ✔ The *heap*, where dynamically allocated variables are allocated from.
- ✔ The *call stack*, where function parameters, local variables, and other function-related information are stored.



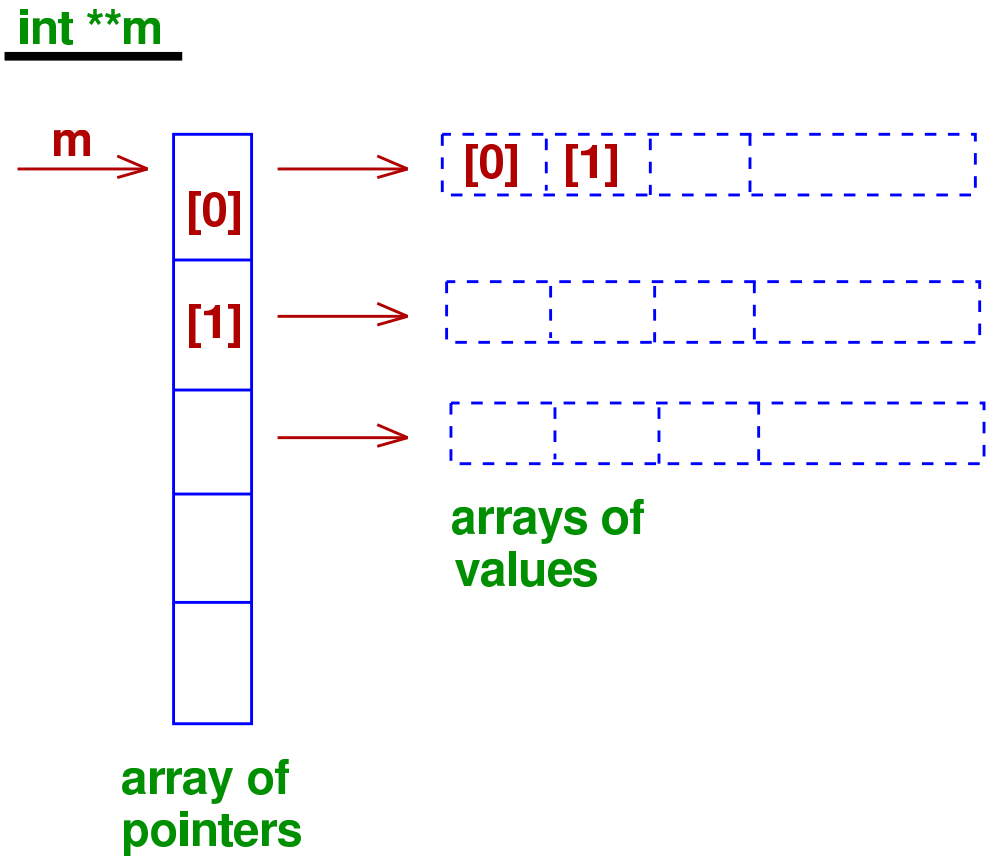
# C++ pointers (cont.)

- ✔ memory allocation in C++: the *new* and *delete* operators

  memory allocated dynamically by the user - **heap memory region**

```
1   //array of strings
2   string s[10];
3   string *s = new string[10]; //allocating memory!!!
4
5   // matrice defined as arrays of arrays (pointer to pointers!)
6   // Define matrice of 10 ROWS * 5 COLS
7   int **m = new int*[10]; // pointer to an array of 10 pointers to integers
8   for (int i=0; i<10; i++) { //ROW arrays
9     m[i] = new int[5]; // m[i] is a pointer to 5 elements 1-dim array
10  }
11
12  // setting values to the 50 allocated memory positions
13  // write values from 0 to 49
14  for (int i=0; i<10; i++) {
15    for (int j=0; j<5; j++) {
16      m[i][j] = i*5 + j;
17    }
18  }
```

# C++ pointers (cont.)






# C++ pointers (cont.)

- ✔ After memory dinamically allocated in the program through the *new* operator we shall at the end of the program free the memory with the *delete* operator

```
1  // accessing and print sequential elements in memory
2  for (int i=0; i<10; i++) {
3    int *p = m[i]; //retrieve address of 1st element of array
4    for (int j=0; j<5; j++) { //printing array elements by incrementing pointer
5      cout << *p << " " << flush;
6      p++;
7    }
8  }
9  cout << endl;
10
11 //free arrays memory
12 delete[] s;
13 for (int i=0; i<10; i++) { delete[] m[i]; }
```

# 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
- ✔ That's why "header"(.h) files are coming at the top of the program

```
1
2  #include <cstdlib> // exit()
3  #include <cstdio> // printf
4
5  //function prototyping
6  double factorial(int);
7
8  ////////////////////////////////////////
9  int main() {
10   for (int i=0; i<=20; i++) {
11     printf("factorial(%d)=%12.3e\n",i,factorial(i));
12   }
13   return 0;
14 }
15
16 ////////////////////////////////////////
17 double factorial(int n) {
18   double fact=1.;
19   if (n<0) {exit(1);} //abort prog if n negative
20   for (int count=n; count > 0; --count)
21     fact *= (double)count;
22   return fact;
23 }
```



# 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 $\sim 3$ executable lines)**

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

# C++ inline functions

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

inline int Max(int x, int y) {
  return (x > y)? x : y;
}

// Main function for the program
int main( ) {
   cout << "Max (20,10): " << Max(20,10) << endl;
   return 0;
}
```



# C++ function pointers

- ✔ a function pointer is a variable that points to the *address* of the function

```
// define function
double sum(double a, double b) {
  return a+b;
}

// main
int main() {
  double (*ptsum)(double,double) = NULL; //init
  ptsum = &sum;
  // calling function
  double c = (*ptsum)(2., 4.5);
}
```

# C++ function pointers (cont.)

- passing a function as argument

```
// func sum
double Sum(double a, double b) {
  return a+b;
}

// receive function pointer
void CalcSum(double (*ptsum)(double,double),
             double a, double b, double& c) {
 c = (*ptsum)(a,b);
}

// main
int main() {
  double c = 0.;
  CalcSum(&Sum, 2., 4., c);
  printf("c=%f\n",c);
}
```



# 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 }
```