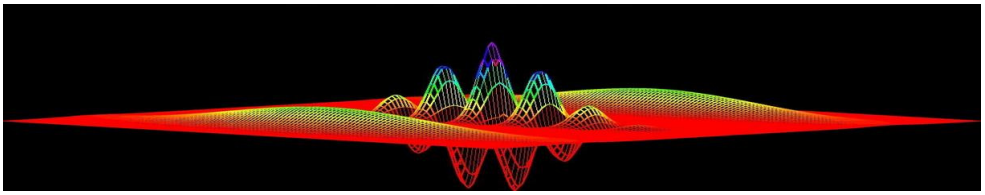
Computational Physics

numerical methods with C++ (and UNIX)



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

C++ Classes and Objects

- ✔ In Object Oriented Programming (OOP) a group is a **class**, a class member is an **object** and a member function implements an **operation**
- ✔ Classes in OOP can be as simple as the set of numbers *int*, *float*, ...
- ✔ The member functions also called **methods** accomplish a broad range of tasks
  - ➜ constructors : default and parametered constructor
  - ➜ accessor member methods : query the objects
  - ➜ mutator member methods : operate and change the object
- ✔ Class members can be **public**, **private** or **protected**

  ➜ public members can be accessed from the user program or user functions
  ➜ private members can only be accessed from class members
  ➜ protected : see inheritance

C++ Classes and Objects (cont.)

- ✔ A member of a class is **private** by default
- ✔ Particular member functions are used to :
  - ➜ create and initialize objects - **constructors**
  - ➜ destroy objects - **destructors**
- ✔ The class declaration needs a semi-colon (**;**) at the end
- ✔ There can be functions, called **friends**, which are not members of the class but have access to private members of the class

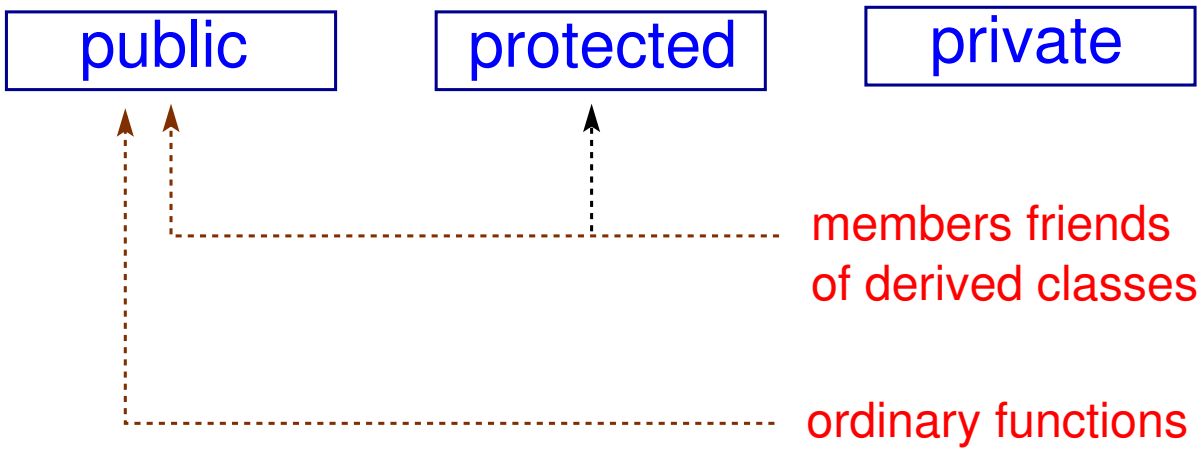
  friend functions can be declared on the private or public sector of the class

```
friend double function();
```

- ✔ Member functions **inline** need to be defined (coded) inside a class declaration (why ? compiler needs to know it...cannot be in a library !)
- ✔ The **struct** data type in C++, is a class with all members **public**

accessing members

kind of members in a class



OOP programming

- A very simple class defining an object *point*
- the *point class* contains two data fields of type *double* : *x* and *y* to store the x and y coordinates of the point object
- **This is not Object Oriented Programming !** In OOP we would like the user to think about the **point** as an object, never dealing directly with its data members !

point class

```
class point {
public:
  double x; //X coordinate
  double y; //Y coordinate
};
```

main.C

```
  point P;
  P.x = 10.;
  P.y = 2.;
```

- The class shall have methods to access the data members (now private)

point class

```
class point {
public:
 double X() const {return x;} // method to access the value of the x coordinate
 double Y() const {return y;} // method to access the value of the Y coordinate
private: //could not be explicitely written (by default they are private)
   double x; //X coordinate
   double y; //Y coordinate
};
```

- *const* declaration implies that a compilation error arises if there is a trial to change the point object being called

Building an object : constructor

- For building an object we simply write :

building point

```
point P;
```

- this declaration makes the C++ compiler to call the *default constructor* of the object that allocates the required memory for the data members of the class and init them

default constructor

```
class point {
  public:
    point() { //default constructor
      x = 0.; y=0.; //init data
    }
};
```

- If no constructor is written, then the C++ compiler invokes its own default constructor and the data members are initialized with random numbers
  write allways your own constructor !

- Build a more sophisticated constructor able to initialize the data members and work also as default constructor
  we just set default values in the arguments !

constructor

```
class point {
  public:
    point(double fx=0, double fy=0) {
      x = fx; y=fy; //init data
    }
};
```

- In the operation above, first memory was allocated for data members and those filled withrandom values and after they were initialized
  this can be done more efficiently with the **initialization list**

constructor with initialization list

```
class point {
 public:
  point(double fx=0, double fy=0) : x(fx),y(fy){;}
};
```

Building an object : copy constructor

- For building an object we can also use another object

building points

```
point P(3.,5.);
point Q(P); //creating Q=(3,5)
point T=P; //creating T=(3,5)
```

- we used the *copy constructor* that made a new object by copying the data members of the object that is passed
- if no *copy constructor* is defined in the class block declaration, then the compiler invokes its *default copy constructor*
- the *copy constructor* is invoked every time an object is passed to a function by value

copy constructor

```
point(const point& p):x(p.x),
                      y(p.y){;}
```

In C++ we can define a *reference* to an existing variable

reference

```
point P;
point& q=P; //reference
```

q is not an independent *point* object but a reference-to-point-object **P**

No copy constructor is used ! -> which means time saving

returning reference to object

```
const point& point::GetObject() {
  //this=pointer to current object
  return *this; /derefence this
}

point P;
point q = P.GetObject();
```


assigning an existing object

- To assign the value of an existing object *Q* to existing point objects *T* and *V* an *assignment operator (=)* must be defined

object assignment

```
point Q, T, V(5.,3.);
Q = T = V; //assignment
```

- Above, the assignment operator shall assign the value of the V object to the current object T, but also return a reference to it (no need to implicit call copy constructor)

copy assignment declaration

```
const point& operator=(const point& p);
```

- **copy assignment implementation**

```
const point& operator=(const point& p) {
  //check if address of current object (this) is tha same of the argument
  if (this != &p) {
    x=p.x;
    y=p.y;
  }
  return *this; //return reference
}
```

Operators

- Other operators can be defined that apply to the current object and update it

+= operator

```
const point& point::operator+=(const point& p) {
  x += p.x;
  y += p.y;
  return *this;
} // adds a point to the current point

point P(1.,2.);
point Q(3.,1.);
Q += P;  // Q = P + Q
```

- negative operator

- operator

```
const point& point::operator-(const point& p) {
  x = -p.x;
  y = -p.y;
  return *this;
}

point P(1.,2.);
point Q = -P;
```


Operators (cont.)

- We can add two points

+ operator

```
point P(1.,2.);
point Q(3.,1.);
point T = P + Q; // P is the current object and Q is the argument

const point point::operator+(const point& p) {
  return point(x+p.x , y+p.y);
} // adds two points
```

Note : cannot be returned a reference to the object because it is local "point" object (disappears when function ends) !

Removing the object : destructor

- The *destructor* of a class its the function called for releasing the memory that the class object allocated

point class destructor

```
class point {
 public:
  ~point(); //destructor
};
```

- if no destructor is defined in the class block, the compiler will invoke its own default destructor
  data is removed from memory in reversed order with respect to the order they appear in the class block
- the compiler default destructor is good enough for objects without data members pointers
  the default destructor would remove only the addresses variables and not the pointed objects !

C++ Classes : an example

Class header (IST.h)

```
#ifndef __IST__
#define __IST__
class IST {
public:
 IST(); // constructor
 ~IST() {;} //destructor
 void SetName(string); // set name
 string GetName() {return name;} // accessor
private:
  string name; float mark;
};
#endif
```

Class implementation (IST.C)

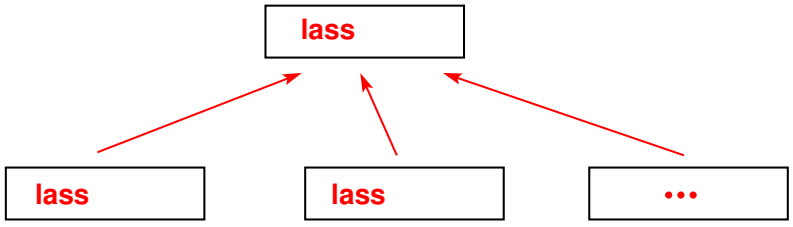
```
#include "IST.h"
IST::IST() { ////// default constructor
  name ="";
  mark=0.0;
}
void IST::SetName(string fname) {
  name = fname;
}
```

using class (test.C)

```
#include "IST.h" //class header

int main() {
  // mem allocated
  IST* pIST = new IST();
  pIST->SetName("Joao N.");
  pIST->SetMark(15.5);

  // vector of pointer objects
  vector<IST*> vIST;
  vIST.push_back(new IST("JJ",15,5));

  //free memory
  delete pIST;
  delete vIST[0];
}
```

C++ classes inheritance

- **MEFT** and **MEEC** are *derived classes* of the *base class* **IST**
- Derived classes inherit all the accessible members of the base class



- The inheritance relationship of two classes is declared in the derived class

```
  class MEFT : public IST {
    public:
            ... //public members
    private:
            ... //private members
};
```

- The keyword **public** specifies the most accessible level for the members inherited from the base class - **all inherited members keep their levels**

  the members of the derived class can access the protected members inherited from the base class but not its private members (invisible members)

C++ classes inheritance (cont.)

- With the keyword **protected**, all *public members of the base class* are inherited as *protected in the derived class*
- the **private** keyword will not give access to the base class members from the derived class

```
\begin{Verbatim}[frame=single]
  class MEFT : protected IST {...};
  class MEFT : private IST {...};
```

- If no access level is specified for the inheritance, the compiler assumes *private* for classes declared with keyword *class* and *public* for those declared as *struct*
- A derived class (public access keyword) inherits every member of a base class except :
  - ➜ its constructors and destructor
  - ➜ its assignment operator members (=)
  - ➜ its friends
  - ➜ its private members
- Nevertheless, the derived class constructor call the default constructor of the base class (the one without arguments) which must exist
  - ➜ calling a different constructor is possible :

```
Derived_Construtor(parameters) : Base_Constructor(parameters) {...};
```

class inheritance : virtual functions

- *Virtual functions* can be declared in a base class with the keyword *virtual* and may be redefined (overriden) in each derived class when necessary
- *Virtual functions* will have the **same name and same set of argument types in both base class and derived class**, but they will perform different actions

```
class Base {
  public:
   //virtual function declaration
   virtual void Function(double);
};

class Derived: public Base {
  public:
   //objects Derived will use this function
   void Function (double);
};
```


class inheritance : abstract classes

- A virtual function declared in a base class can eventually stay undefined due to lack of information - it will be called a *pure virtual function*

pure virtual function

```
class Base {
  public:
   //pure virtual function
   virtual void Function(double) = 0;
};
```

- A class with one or more pure virtual functions is called an *abstract class*
- **No objects of an abstract class can be created**
- A pure virtual function that is not defined in a derived class remains a pure virtual function and the derived class is also an abstract class

C++ classes inheritance (cont.)

Base class header (IST.h)

```
#ifndef __IST__
#define __IST__
class IST {
public:
 IST(); // NEEDED default constructor
 IST(string, float); // constructor
 ~IST() {;} //destructor
 void SetName(string);
 string GetName();
 virtual void SetBranch(string)=0;
protected:
  string name;
  float mark;
};
#endif
```

Derived class header (MEFT.h)

```
#ifndef __MEFT__
#define __MEFT__
class MEFT : public IST {
public:
 MEFT(string, float, string); //constr
 ~MEFT() {;} //destructor
 void SetBranch(string);
 string GetBranch();
protected:
  string branch; //curso
};
#endif
```

Base class code (IST.C)

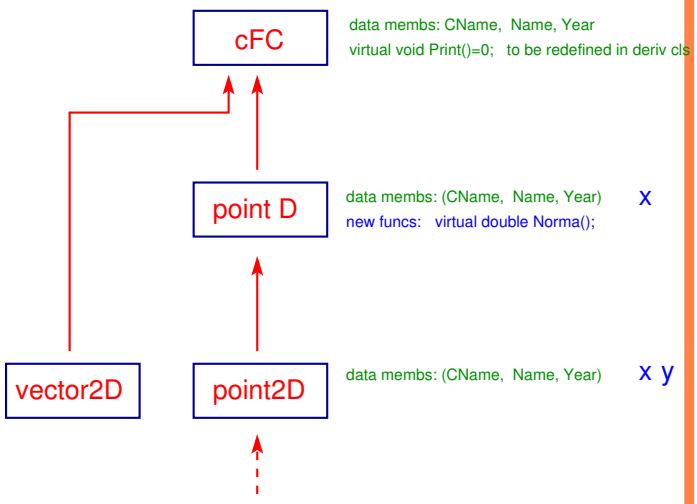
```
#include "IST.h"
IST::IST(string fname, float fmark) : name(fname), mark(fmark) {;} // ... code
```

Derived class code (MEFT.C)

```
#include "MEFT.h"
MEFT::MEFT(string fname, float fmark) : IST(fname, fmark) {;}
void MEFT::SetBranch(string fbranch) {branch = fbranch;} // ... code
```


An inheritance scheme for Fis Comp

- ✔ Let's define a base class that should define basic information common to all classes to be developed - **cFC class**
  - ➜ the group name *(string)*
  - ➜ the scholar year *(string)*
  - ➜ the class name *(string)*
  - ➜ virtual functions supposed to be redefined in derived classes



- ✔ The classes that derive from *cFC class* will inherit all members of base class and will :
  - ➜ provide replacements for virtual's funcs
  - ➜ add new data members
  - ➜ add new functions
- ✔ A derived class can be a base of another derived class

cFC class : header file

Class header (cFC.h)

```
#ifndef __cFC__
#define __cFC__
#include <string>
#include <iostream>
using namespace std;
class cFC {
 public:
  cFC() {groupName="; Year="; ClassName="; }
  cFC(string fg, string fy) : groupName(fg), Year(fy) {};
  string GetGroupName();
  string GetYear();
  void PrintGroupId();
  virtual void Print() = 0; //generic print to be implemented in every derived class
  void SetClassName(string fc) {ClassName = fc;}
  string GetClassName() {return ClassName;}
  void PrintClassName() {cout << ``Class Name = `` << ClassName << endl;}
 private:
  string groupName;
  string Year;
  string ClassName; //+...(nome do trabalho, ...)
};
#endif
```


cFC class : code

Class implementation (cFC.C)

```
#include <iostream>
using namespace std;
#include "cFC.h"

string cFC::GetGroupName() {
  return groupName;
}
string cFC::GetYear() {
  return Year;
}
void cFC::PrintGroupId() {
  cout << "group Name   = " << groupName << endl;
  cout << "Scholar year = " << Year << endl;
}
```

point1D class : header file

Let's define a class to manipulate one-dimensional points : **Class header (point1D.h)**

```
#ifndef __point1D__
#define __point1D__
#include "cFC.h"
#include "point1D.h"

  class point1D : public cFC { // 1D points

  public:
    point1D(double fx=0.) : cFC("A01","2014-15"), x(fx) {
      SetClassName("point1D"); } // default constructor (inlined)
    void move(double); //move to new position
    void move(point1D); //move to new position
    void Print(); //print
    virtual double Norma(); //calculate modulo

  protected:
    double x; // x coordinate
  };
#endif
```


class : comments

cFC

- ✔ abstract class due to pure virtual function *Print()*
- ✔ reminder : abstract class cannot be instatiated by itself !
- ✔ the virtual function must be defined by the derived classes

point1D

- ✔ class has protected members *x*, which means visible to derived classes members
- ✔ constructor code is implemented inside header file
  - ➜ *inlined* constructor
  - ➜ shows that implementation can follow declaration
- ✔ There is *default constructor (constructor with no arguments)*
- ✔ destructor is not needed because there is no space allocated on *heap* by the class
- ✔ overloading of member functions *move()*

point1D class : code implementation

Class code (point1D.C)

```
#include <iostream>
using namespace std;
#include "point1D.h"

void point1D::move(double fx) {x=fx;}

void point1D::move(point1D p) {x=p.x;}

void point1D::Print() {
 PrintClassName();
 cout << ``[point1D] x='' << x << endl;
}

double point1D::Norma() { return x;}
```


point2D class

main program (main.C) YOU HAVE TO TRY IT!!!!

```
class point2D : public point1D {
  public:
   point2D(double fx, double fy) : point1D(fx), y(fy) {;}
   ...
  private:
   double y; // y coordinate
};
```

main program (main.C) YOU HAVE TO TRY IT!!!!

```
#include "point2D.h"
int main() {
   point2D a; // try this...! which constructor is being used?
   a.Dump();

   point2D b(0,0);  b.Dump();

   point2D c(5,2);
   b.move(c); //b=(5,2)
   b.Dump();
   double d = Norma(b);
}
```

point2D class (cont.)

- ✔ You are going to have a compiler error due to the fact you are trying to instantiate a *point2D* using the default constructor (NOT IMPLEMENTED !)
- ✔ Implementation of a default constructor

```
point2D() {x=0; y=0;}
```

- ✔ You can define a much more generic constructor that is a default constructor (no arguments needed) and also accepts arguments

```
point2D(double fx=0, double fy=0) : x(fx), y(fy) {;}
```

Example of use of the different constructors

```
point2D a; // (0,0)
point2D b(5); // (5,0)
point2D b(5,2); // (5,2)
```


class vector2D

Let's make a class **vector2D** making use of the class **point2D** before defined ; it will include two point2D data members dynamically allocated that will require the user to define copy's constructor and assignment

a possible class definition with two points (vector2D.h)

```
class vector2D : public cFC {
  public:
   vector2D(point2D pf, point2D pi) : cFC("A01", "2014-15"), Pf(pf), Pi(pi) { cout <<
   vector2D(point2D pf) : cFC("A01", "2014-15"), Pf(pf), Pi() { cout << "point2D const
  private:
   point2D Pi; //initial point
   point2D Pf; //final
};
```

class definition with a point2D pointer (vector2D.h)

```
class vector2D {
  public:
   vector2D(point2D pf, point2D pi);
   vector2D(point2D pf);
  private:
   point2D *P; //pointer
};
```

class implementation (vector2D.C)

```
vector2D::vector2D(point2D p2, point2D p1) {
  P = new point2D[2];
  P[0] = p1; P[1] = p2; }
vector2D::vector2D(point2D pf) {
  P = new point2D[2];
  P[0] = point2D(); //default constr
  P[1] = pf; }
```

class vector2D (cont.)

vector2D class : copy and assignment constructor declarations (vector2D.h)

```
class vector2D {
  public:
   vector2D(const vector2D&); //copy constructor
   vector2D& operator=(const vector2D&); //copy assignment
   ...
};
```

vector2D class : copy and assignment constructor implementation (vector2D.C)

```
   vector2D::vector2D(const vector2D& t) {//copy constructor
    P = new point2D[2]; // array with two points created
    P[0] = t.P[0];
    P[1] = t.P[1];
   }
   vector2D& vector2D::operator=(const vector2D& t) {//copy assignment
    if (this != &t) { //this is a const pointer to current object (member func invoked
       P[0] = t.P[0];
       P[1] = t.P[1];
    }
    return *this;
   }
```


class inheritance : virtual destructors

- ✔ A virtual destructor is a destructor that is also a virtual function
- ✔ The virtual destructor can ensure a proper cleanup of an object

```
class Base {
  public:
   virtual ~Base();
};

class Derived: public Base {
  public:
  ~Derived();
};
```

C++ classes polymorphism

- **pointers to base class**
  The class inheritance allows the polymorphic characteristic that a pointer to a derived class is type-compatible with a pointer to its base class
  A class method argument can use generically a pointer to the base class for passing any derived class object (only members of the base class are available to base class pointer)
  To recover the original object a cast is needed :

```
derived_class *p = (*derived_class) base_class_pointer;
```

- **virtual functions**
  A class that declares or inherits a virtual function is called a polymorphic class.
  A virtual method is a member function that can be redefined in a derived class.
  If the virtual member is =0 we are in presence of an abstract base class that cannot be instatiated by itself !

```
Example: virtual string GetBranch();
Example: virtual string GetBranch() = 0; //pure virtual function
```


C++ class static members

- *class static member* is a member of a class and not of the objects of the class.
  there will be exactly one copy of the static member per class
- a function that needs to have access to members of a class but need not to be invoked for a particular object, is called a *static member function*

```
class vector2D {
  private:
   pointer2D *P;
   static point2D InitPoint; //init point defined for all objects
  public:
   static void SetInitPoint(const point2D& );
};
```

Note : private static data members cannot be accessed publicly (only from class members)

- Initializing static variable (in vector2D.C)

```
//init static variable with (0,0)
  point2D vector2D::InitPoint=point2D();
// implementation of the class code
  vector2D::vector2D(point2D p2, point2D p1) {
  ...
```

C++ class static members (cont.)

- calling static function from main.C

```
#include "vector2D.h"
#include "point2D.h"

  int main() {
    //call static function and set static variable to (1,1)
    vector2D::SetInitPoint(point2D(1,1));
  }
```


C++ class static methods

- *Static methods* can be implemented in classes in order to provide functions within a class scope that does not need any private member the class works as a repository of functions that have to be called from the user-function with the scope operator

```
class USERtools {
public:
  // computes maximum value of array
  double MaxValue(double*);
};

#include "USERtools.h"
int main() {
  double p[] = {0.23, 053, 2.3, 5.6, 7.};
  double result = USERtools::MaxValue(p);
}
```