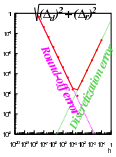
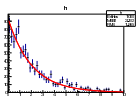
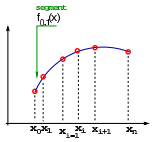
Computational Physics

numerical methods with C++ (and UNIX)

2018-19



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

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)

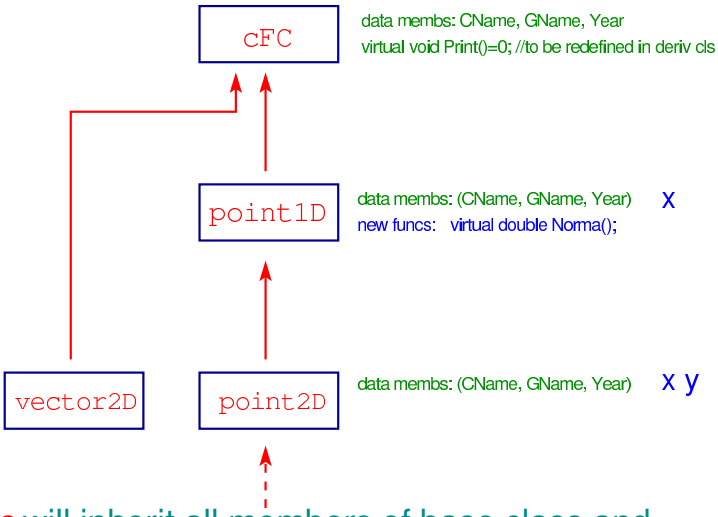
```
#include "IST.h"
IST::IST(string fname, float fmark) : name(fname), mark(fmark) {;} // ... code
```

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

point2D.h

```
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 << "point2D constructor/1" << endl;}
   vector2D(point2D pf) : cFC("A01", "2014-15"), Pf(pf), Pi() {
     cout << "point2D constructor/2" << endl;}
  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 constructor
  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

- ✔ When called, the destructor of the derived class calls also the base class destructor
- ✔ Nevertheless, there are situations where that do not happen
  - ► It is better to have a base class 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() {
   cout << "Base destructor called"
        << endl;
   }
};

class Derived: public Base {
  public:
  ~Derived() {
    cout << "Derived destructor called"
         << endl;
  }
};
```

```
// both destructors called (derived, base)
Derived *D1 = new Derived();
delete D1;

// only Base destructor is called if
// no virtual destructor on base class
Base *D2 = new Derived();
delete D2;
```


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 instantiated 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
    static double MaxValue(double*);
};

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


C++ inheritance example

Playing with inheritance and polymorphism: C -> B -> A

class A

```
#define DEBUG
#undef DEBUG

#include <iostream>
using namespace std;
#include <cstdio>
#include <cmath>

class A {
  public:
    A(float fx=0.0) : x(fx) {cout << __PRETTY_FUNCTION__ << "Constructor A called" << endl;}
    virtual void Print() {cout << __PRETTY_FUNCTION__ << "print A (x=" << x << ")" << endl;}
    virtual ~A() { cout << __PRETTY_FUNCTION__ << "destructor A called" << endl;}
  protected:
    float x;
};
}
```

C++ inheritance example

class B

```
class B: public A {
  public:
    //B(float fx=0.0) : x(fx) { // NOT CORRECT
    //B(float fx=0.0) { x=fx; // CORRECT but not fast
      B(float fx=0.0) : A(fx) { // CORRECT FAST
        cout << __PRETTY_FUNCTION__ << "Constructor B called" << endl;}
      ~B() {
        cout << __PRETTY_FUNCTION__ << "destructor B called" << endl;}
      void Print() {
        cout << __PRETTY_FUNCTION__ << " print B (x=" << x << ")" << endl;}
      virtual float Norma() {return fabs(x);}
};
```

class B* inherits from *class A

***Print()* method redefined**

***Norma()* method defined as virtual**