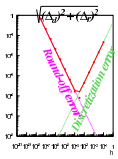
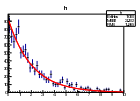
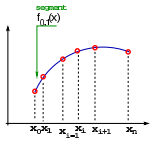
Computational Physics

numerical methods with C++ (and UNIX)

2018-19



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

C++ Operators overloading

✔ commonly overloaded operators on user-defined classes

assignment operator	=
binary arithmetic operators	+ − *
compound assignment operators	+= −= *=
comparison operators	== !=
unary operators	++ −− − !

Adding two points

binary operator +

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

point point::operator+(const point& A) {
  return point(x+A.x , y+A.y);
  // in case we had pointers on private members
  return point(*x+*(A.x), *y+*(A.y));
} // adds two points
```

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

C++ compound assignment operators

- Compound assignment operators are destructive operators; they update or replace the values on left-hand side of the assignment
  they 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
```

C++ unary operators

- they apply to the current object modifying or not their values

unary operator -

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

point P(1.,2.);
point Q = -P; // P.operator-() && copy constructor

point E;
E = -P; // P.operator-() && assignment operator
```


C++ comparison operators

comparison operator ==

```
bool point::operator==(const point& A) {
  if (*x == *(A.x) && *y == *(A.y)) return true;
}

point P(1.,2.);
point Q = P;
if (Q==P) cout << "similar points!" << endl;
```

Creating class objects

✔ Now that we understood the constructor role we can build objects and refer to the public available functions

locally

local object point

```
// make a point
point P(1.,2.);
P.Print(); //print point
P.X(); // look to x coo
P.Y(); // look to x coo
```

dynamically

local object point

```
// make a pointer to a new object
// constructor called
point *p = new point(1.,2.);
//print point (note the ->)
p->Print();
p->X(); //look to x coord
p->Y(); //look to y coord
```

class point

```
class point {
public:
 //methods publically visible

 point(double fx=0, fy=0):x(fx), y(fy){;} //constr
 point(const point& p):x(p.x),y(p.y){;} //copy constr
 point& operator=(const point& p); //assignment
 point& operator+=(const point& p); //+=
 point& operator-(const point& p); //-
 point operator+(const point& p); //+

 double X() const {return x;} // access the x coord
 double Y() const {return y;} // access the y coord
 void SetX (double); // set the x coord
 void SetY (double); // set the y coord
 void Print(); // print point

private:
  double x; //X coordinate
  double y; //Y coordinate
};

point A; point B(A); //copy constructor
point A=B; //copy constructor
A=B; //A.operator=(B), assignment operator
A+=B; //A.operator+=(B),
A=A+B; //A.operator=(A.operator+(B))
point C = A+B; //A.operator+(B) && copy constructor called
point D = A-B; //A.operator-(B) && copy constructor called
```


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)

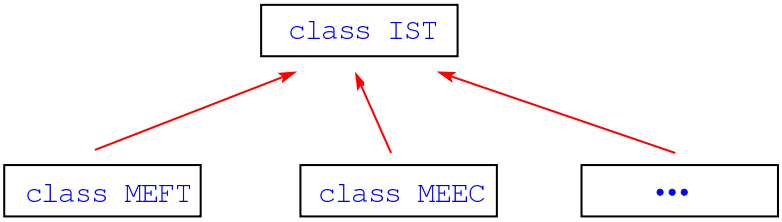
```
#include "IST.h"
IST::IST() { ////// default constructor
  name ="";
  mark=0.0;
}
void IST::SetName(string fname) {
  name = fname;
}
```

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

```
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