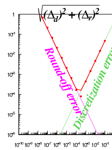
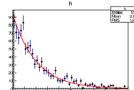
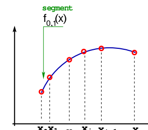
Computational Physics

numerical methods with C++ (and UNIX)

2018-19







Fernando Barao

Instituto Superior Tecnico, Dep. Fisica

email: fernando.barao@tecnico.ulisboa.pt

C++ Classes 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

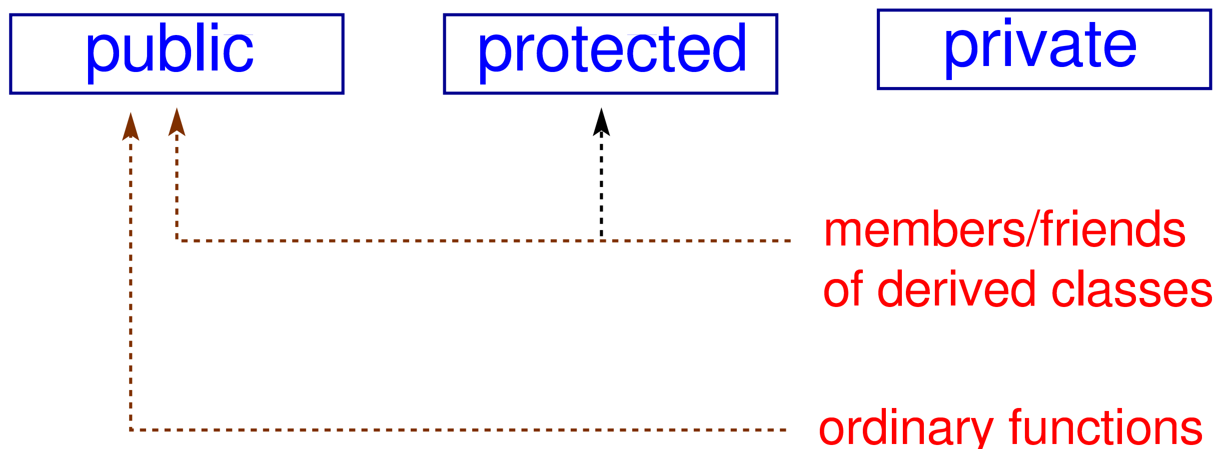
  friend functions can be declared on the private or public sector of the class

```
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 is similar to do: T.operator=(V);
```

- 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 the current object (this) is the same of the argument
  if (this != &p) {
    x=p.x;
    y=p.y;
  }
  return *this; //return reference
}
```

move constructor

✔ Content of a source object is transferred to a destination object; the source loses its content.
this happens to *unnamed objects*, objects that are temporary by nature and thus haven't yet a name!

✔ the move constructor is called when an object is initialized on construction using an unnamed temporary

using move constructor

```
point P = fn(); // move constructor called, fn() returns an object point
point P = point(); // move assignment called
point A; point P = A; // copy constructor called
```

move constructor

```
class point {
  double *x, *y;
  public:
  // constructor
  point (double fx=0., double fy=0.) {
    x = new double(fx); //init to zero
    y = new double(fy);
  }
  // destructor
  ~point() {delete x; delete y;}
  // move constructor (memory already allocated is transferred)
  point (point&& P) : x(P.x), y(P.y) {P.x = nullptr; P.y=nullptr;}
};
```

move assignment

move assignment

```
class point {
  double *x, *y;
  public:
  // constructor
  // initialize members though init list (fastest way)
  point (double fx=0., double fy=0.) : x(new double(fx)), y(new double(fy)){;
  // destructor
  ~point() {delete x; delete y;}
  // move assignment (memory already allocated is transferred)
  point& operator= (point&& P) {
    delete x;
    delete y;
    x = P.x;
    y = P.y;
    P.x = nullptr;
    P.y = nullptr;
    return *this;
  }
};
```

class special member functions

✔ There are some class member functions that can be **implicitely defined** under certain conditions!

default constructor	C::C()	if no other constructors
destructor	C::~C()	if no destructor
copy constructor	C::C(const C&)	if no move constructor and no move assignment
copy assignment	C& operator= (const C&)	if no move constructor and no move assignment
move constructor	C::C (C&&)	if no destructor, no copy constructor and no copy nor move assignment
move assignment	C& operator= (C&&)	if no destructor, no copy constructor and no copy nor move assignment