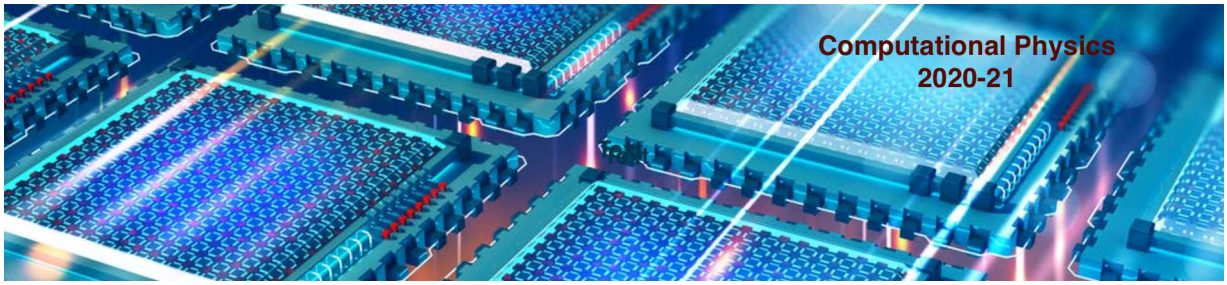
Computational Physics

numerical methods with C++ (and UNIX)

2020-21



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

Computational Physics
Classes and Objects
OOP programming

Fernando Barao, Phys Department IST (Lisbon)

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 constructors
  - **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.)

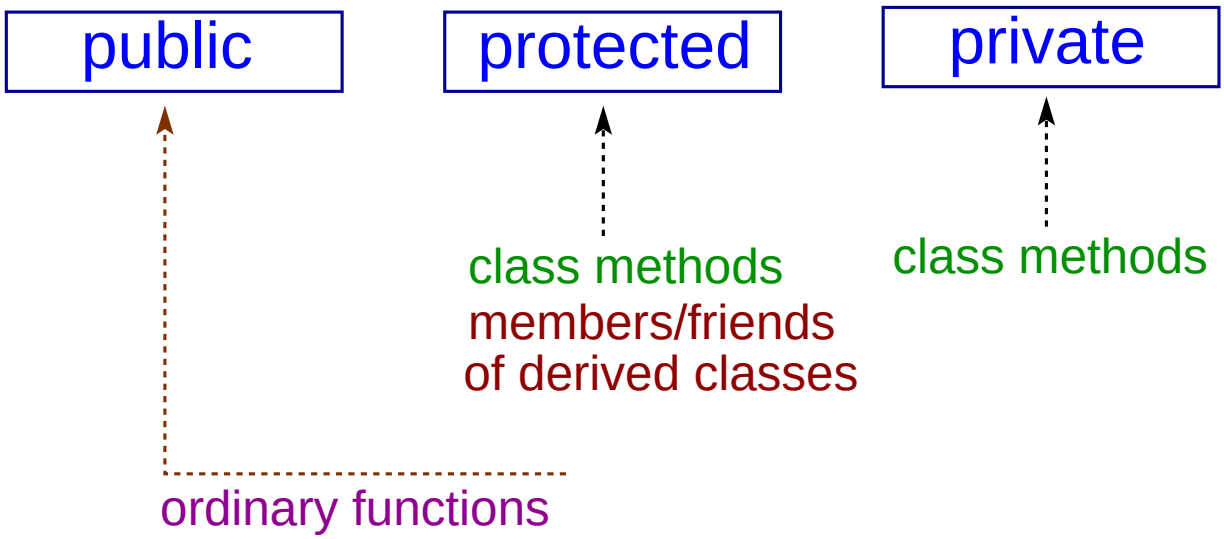
- A member of a class is **private** by default
- Particular member functions are used to:
  - create and initialize objects - **constructors**
  - destroy objects - **destructors**
- The class declaration needs a semi-colon (**;**) at the end
- There can be functions, called **friends**, which **are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

  friend functions are declared on the private or public sector of the class

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

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

kind of members in a class: access



OOP programming

- ✔ A very simple class defining an object *point*
- ✔ *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, **1)** memory was allocated for data members that were filled with random values and **2)** members 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: example

point (2D) class header

```
1  #ifndef __point__
2  #define __point__
3
4  class point {
5   public:
6
7    // constructor, destructor
8    point(float x=0., float y=0.);
9    ~point(); // destructor
10
11   // other
12   void Dump() const;
13
14  private:
15   // point coordinates
16   float* coo;
17 };
18 #endif
```

main program

```
1  #include "point.h"
2
3  int main() {
4    printf("____ constructor\n" );
5    point P;
6    point P(0.25,3.1);
7    printf("____ ending\n" );
8  }
```

output

```
_________ constructor
[point::point(float, float)] constructor...
[void point::Dump() const] (0x7ffee48bd460) 0.000000 0.000000
[point::point(float, float)] constructor...
[void point::Dump() const] (0x7ffee48bd448) 0.234000 1.093000

_________ ending
[point::~point()] destructor... (0x7ffee48bd448)
[point::~point()] destructor... (0x7ffee48bd460)
```

notice reverse order of objects destruction wrt creation

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; //de-reference this pointer
}

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


Default copying: example

point (2D) class header

```
1  #ifndef __point__
2  #define __point__
3
4  class point {
5   public:
6    // constructor, destructor
7    point(float x=0., float y=0.);
8    point(const point&) = default;
9    ~point() = default; //destructor
10
11   // What is a good destructor?
12   // something that shall erase
13   // allocated memory
14   // => delete [] coo
15
16   private:
17   // point coordinates
18   float* coo;
19 };
```

main program

```
1  #include "point.h"
2  int main() {
3    point A(0.25,3.1);
4    point B(A);
5    B.Dump();
6    point C = A;
7    C.Dump();
8  }
```

output

```
[pointS::pointS(float, float)] constructor...
[void pointS::Dump() const] 0.234000 1.093000
obj pointer=0x7ffee298d460 | C-array pointer=0x7fd557c058a0
[void pointS::Dump() const] 0.234000 1.093000
obj pointer=0x7ffee298d458 | C-array pointer=0x7fd557c058a0
[void pointS::Dump() const] 0.234000 1.093000
obj pointer=0x7ffee298d450 | C-array pointer=0x7fd557c058a0

_________ ending
```

1. Is default copy constructor correct?
   NO! Object not replicated!!!
2. Is default destructor correct?
   NO! MEMORY LEAKAGE!!!!!

copy assignment: operator=

- 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 object reference
}
```


move semantics

move semantics is a concept introduced by C++11 revision that allows the compiler to replace *copy concept* by *move concept*: a temporary object (i.e., that will disappear) is moved to an addressable object

- **lvalues**: (left hand side value) its address is recoverable (& operator)

lvalues

```
float x = 10.;  // x has a memory address recoverable, so it's an lvalue
                // 10. is an rvalue
float *px = &x; // address of x (x can be changed)
}
```

- **rvalues**: (right hand side value) temporary object as for instance a variable returned by value on a function

rvalues

```
float x = 10.;  // x has a memory address recoverable, so it's an lvalue
float y = (x+1.); // (x+1) is an rvalue
string(''text''); // temporary object
```

- **rvalues references:** reference to r value variable (&& operator)

moving a large object (i.e., vampirize it) is much more code efficient than **allocate memory + copy**

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
```

point class with 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;}
  (...)
```

```
class point {
  double *x, *y;
  public:
  (...)
  // move constructor
  // memory already allocated is moved
  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 moved)
  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

C++ Operators overloading

✔ commonly overloaded operators on user-defined classes

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

Adding two objects

```
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);
```

binary operator +

```
// adds two points
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));
}
```

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

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

+= operator

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

C++ unary operators

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

```
1 point P(1.,2.);
2 point Q = -P; // P.operator-() && copy constructor
3
4 point E;
5 E = -P; // P.operator-() && copy assignment
```

unary operator -

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


C++ comparison operators

```
1 point P(1.,2.);
2 point Q = P;
3 if (Q==P) {
4   cout << "similar points!" << endl;
5 }
```

comparison operator ==

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

91-1