Object Oriented Programming

3 Classes & Objects

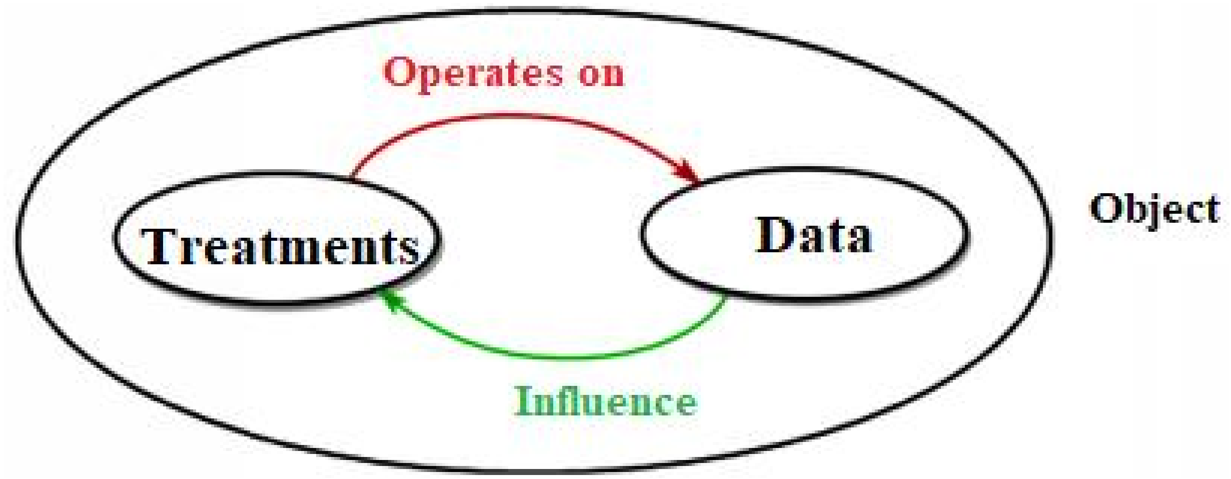
-

Object Concept

OOP allows to improve :

- Robustness (change of specification)
- Modularity
- Readability

=> **The maintenance**



OOP: Four basic concepts

One of the main objectives of the notion of object: to organize complex programs thanks to the notions of:

- encapsulation
- abstraction
- inheritance
- and polymorphism

Class Concept

A class is a template from which we can generate a set of objects sharing common attributes and methods. The objects belonging to it are the instances of this class i.e. Instantiation is the creation of an object of a class.

Class declaration

In C++, a class is defined using the keyword "class" followed by its name. This name usually begins with a capital letter. Following this, the data members and methods are declared between two braces ({}). The definition of the class must end with a semicolon:

Syntax:

```
class Name
{
 // Member Data and Member Functions
};
```

Class declaration (Example)

Let's take a concrete and simple example: writing a Point class:

- In C, we would have made a structure like this:

```
struct Point
{
 int x;
 int y;
};
```

The previous statement works perfectly in C++. But we would like to add functions that are strongly related to this data, such as displaying a point, moving it, etc.

Class declaration (Example)

```
#include <iostream>
class Point
{ public:  // The attributes
 int x;
 int y;
  // The methods
 void display()
 { cout <<x<<”,”<<y<<endl; }
void place (int a ,int b)
 { x=a;
 y=b;
 }
 void move (int a, int b)
 { x += a;
 y += b;
 }
};
```

Use of Class (Class Instance)

- A class being defined, it is possible to create objects of this type.

- We declare an "object" of a given class type by preceding its name with that of the class.

```
Point p1, p2;
```

Accessing an object's member

Static declaration case

```
#include <iostream>
#include "Point.cpp"
int main()
{
 Point p;
 p.x=10;
 p.y=20;
 p.display();
 p.place(1,5);
 p.display();
return 0;
}
```

Accessing an object's member

Dynamic allocation case

```
#include <iostream>
#include "Point.cpp"
int main()
{
Point *p=new Point;
 p->x=10;
 p->y=20;
 p->display();
 p->place(1,5);
 p->display();
return 0;
}
```

Scope Resolution Operators (1/3)

The methods of the Point class are implemented in the class itself. This works very well, but of course becomes quite heavy when the code is longer. This is why it is **better** to place the declaration only, within the class.

Scope Resolution Operators (2/3)

```
#include <iostream>
class Point
{ public :
 int x;
 int y;
 void placer(int a, int b);
 void deplace(int a, int b);
 void affiche();
};
void Point::placer(int a, int b)
{ x = a;
 y = b;
}
void Point::deplace(int a, int b)
{ x += a;
 y += b;
}
```

```
void Point::affiche()
{
 cout << x << ", " << y << endl;
}
int main()
{ Point p;
 p.placer (3,4);
 p.affiche();
 p.deplace(4,6);
 p.affiche();
}
```

Scope Resolution Operators (3/3)

We notice the presence of the "Point::" (:: scope resolution operator) which means that the function is in fact a method of the Point class. The rest is completely identical.

The only difference between the two ways comes from the fact that we say that the functions of the first (whose implementation is done in the class), are "inline".

Objects passed as arguments to a member function

We want to compare two points, in order to know if they are equal. To do this, we will implement a method "Coincide" that returns "true" when the coordinates of the two points are equal.

Exercise:

Write the code of the Member funtion (the method) Coincide

Objects passed as arguments to a member function

```
class Point
{ public :
 int x;
 int y;
 bool Coincide (Point &p)
 { if( (p.x==x) && (p.y==y) )
 return true;
 else
 return false;
 }
} ;
```

Objects passed as arguments to a member function

```
#include <iostream>
class Point
{ public :
 int x;
 int y;
 bool Coincide(Point &p)
 { if( (p.x==x) && (p.y==y) )
 return true;
 else
 return false;
 }
};
```

```
int main(){
Point p;
Point pp;
pp.x=0;pp.y=1;
p.x=0; p.y=1;
if( p.Coincide(pp) )
cout << "p et pp coincide !"
<< endl;
}
```

Method whose type is the same as the class (1/2)

Suppose we need to determine the midpoint between two points?

Solution: We can add to the point class a method that provides this point, this method accepts as an argument a point P and it must produce the midpoint between P and the point represented by the class where the result (return type) is a point.

Exercise: Write the method midpoint?

Method whose type is the same as the class (2/2)

```
class Point
{ public :
int x;
int y;
void init(int a, int b)
{ x=a;
 y=b;
}
Point midpoint(Point &p)
{ Point Mid;
 Mid.x=(x+p.x)/2 ;
 Mid.y=(y+p.y)/2 ;
 return Mid ;
} };
```

```
int main()
{
Point P1, P2, M;
P1.init(1,3);
P2.init(4,4);
M=P1.midpoint(P2);
//Or M=P2.midpoint(P1);
}
```

Data members of type Structure (struct) (1/3)

- We want to define a class that allows us to create ‘persons’; a person is characterized by their **name**, **first name**, **NI number** and **date of birth**.

- However, the **date of birth** is a field composed of day, month and year, so it must be grouped under a single name using a Date structure:

  ✓ The structure is defined outside the class
  ✓ The structure is defined inside the class

Data members of type Structure (struct) (2/3)

```
struct Date
{ int d,m,y;
} ;

class Person
{ public :
 char Name[50] ;
 char FirstN[50];
 int NIN ;
 Date DB
//...
};
```

```
int main()
{ Person P ;
 Date D ;
 D . j = 1   ;   D . m = 1   ;
D.a=2008 ;
 strcpy(P.Name, "ali") ;
 P.DB=D ;
};
```

Data members of type Structure (struct) (3/3)

```
class Person
{
struct Date
{ int d,m,y;
} ;
 public :
 char Name[50] ;
 char FirstN[50];
 int NIN ;
 Date DB
//...
};
```

```
int main()
{ Person P ;
 Date D ; Compilation Error
 D.j=1 ; D.m=1 ; D.a=2008 ;
 strcpy(P.Name, "ali") ;
 P.DB=D ;
};
```

Classes composed of other classes (1/4)

Here we are talking about member objects: when an attribute of the class is itself of type another class.

```
class Date
{ public:
 int d, m, y ;
 void display()
 { cout<<d<< "/"<<m<<"/"<<y<<endl ;}
 void init(int dd, int mm, int yy)
 { d=dd ; m=mm ; y=yy ; }
};
```

Classes composed of other classes (2/4)

```
class Person
{ public :
 char Name[50] ;
 char FirstN[50];
 int NIN ;
 Date DB ; // DB is an instance of  Date
 void display()
 {
 cout<< "Name="<<Name<<endl ;
 cout<< "Firste Name="<<FirstN<<endl ;
 cout<< "National Identity Number="<<NIN<<endl ;
 cout<< "Birthday= " ;
 DB.display() ;  }//... };
```

Classes composed of other classes (3/4)

We can declare the class Date in Person. In this case, Date can only be used in the Person class..

```
class Person
{class Date
 {public :
 int d, m,y;
 void display()
 { cout<<d<< "/"<<m<<"/"<<y<<endl ;}
 void init(int dd, int mm, int yy)
 { d=dd ; m=mm ; y=yy ; }
 };
```

Classes composed of other classes (4/4)

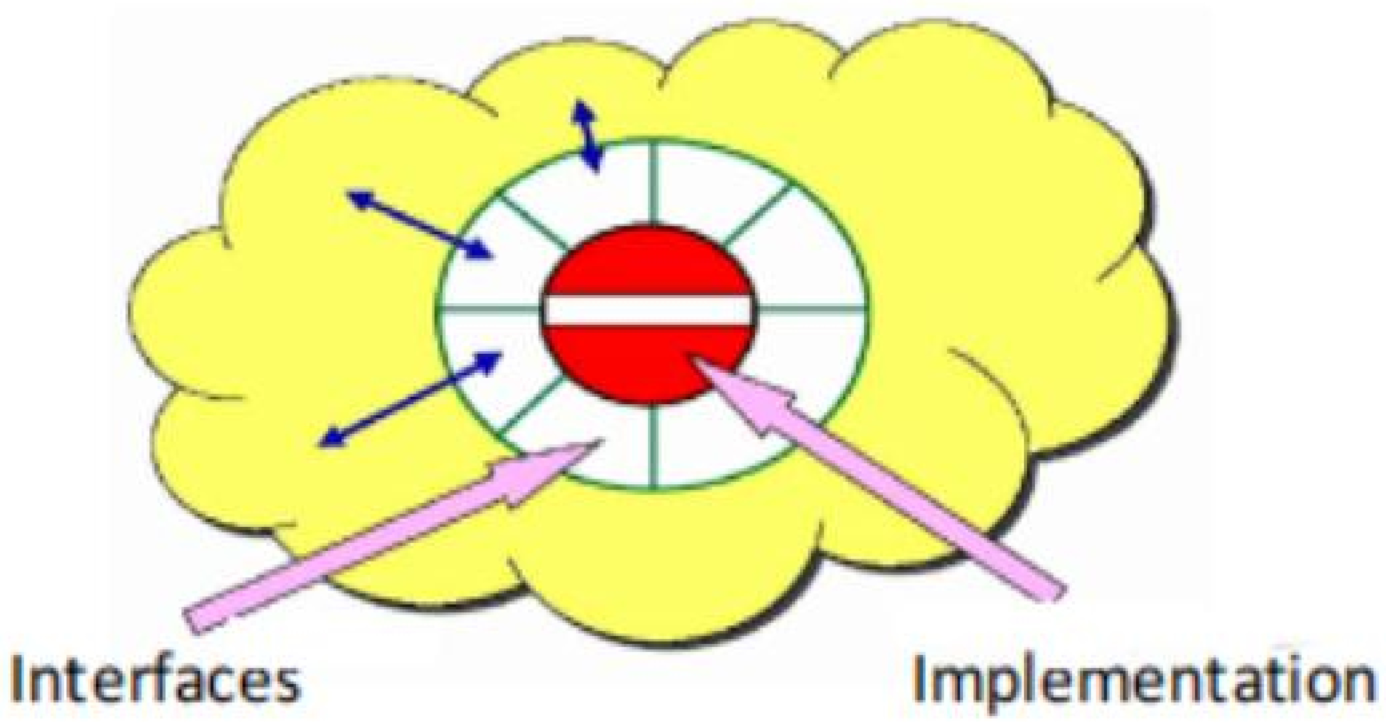
```
/*...................class Person..............*/
 public :
 char Name[50] ;
 char FirstN[50];
 int NIN ;
Date DB ; // DB is an instance of  Date
 void display()
 {
 cout<< "Name="<<Name<<endl ;
 cout<< "Firste Name="<<FirstN<<endl ;
 cout<< "National Identity Number="<<NIN<<endl ;
 cout<< "Birthday= " ;
 DB.display() ;  }//... };
```

Encapsulation (1/2)

- Encapsulation or information hiding consists of separating the external aspects of an object accessible to others, which are called public, from the internal implementation details, made invisible to others, which are called private.

- Technique that isolates the external aspect of the object from its internal aspect=> Data and methods are gathered together. They are **hidden** and **protected**.

Encapsulation (2/2)



Concept of visibility (1/4)

- The visibility of a feature (a member data) determines whether other classes can use it directly. Three levels of visibility are used:
- Public: Member data and functions are accessible throughout the application.
- Protected: Member data and functions are only accessible by member functions of the class and any **derived** classes (see inheritance).
- Private: Member data and functions are only accessible by member functions of the class. By default, members are private.

Concept of visibility (2/4)

```
#include <iostream>
using namespace std ;
class Points
{ int color; //private member
 void color(int pcolor)
 { color = pcolor; }
 public :
 int x, y;
 void display()
 { cout <<this->x<<”,”<<this->y<<endl;// Useless natation with this
 void place (int x , int y)
 { this-> x = x; // Useful notation with this
   this->y = y; } };
```

Concept of visibility (3/4)

```
#include <iostream>
#include "Points.cpp"
using namespace std ;
int main()
{ Points *p=new Points;
 Points *p1=new Points;
 p1->x=1;
 p1->y=15;
```

Concept of visibility (4/4)

/*............. class Point................*/

p1->color=1; // Compilation error because color is a private member

p1->color(10); // Compilation error because color() is a private method }

The keywords public, private and protected can appear multiple times in the declaration of the same class.

Access methods and modification methods (the getters and setters)

- Accessors (the getters): get<Attribute Name>() is a method of reading a private attribute.

- Modifiers (the setters): set<AttributeName>(.....) is a method to modify a private attribute.

Access methods and modification methods (the getters and setters)

```
class Rectangle
{public:
 int Perimeter()
 { return 2*(W+H); }
 int surface()
 { return W*H; }

// The accessors

 int getW()
 { return W; }
 int getH()
 { return H; }
```

```
// The modifiers

 void setW(int newW)
 { W=newW; }

 void setH(int newH)
 { H=newH; }

private :
 int W;
 int H;
};
```

Initialization and Constructor (1/7)

Initialization is the process of assigning values to variables when declaring them.

How can we initialize an object?

What mechanism does object-oriented programming provide for initializing objects?

Initialization and Constructor (2/7)

Initialization: Assigning values during declaration

```
class account
{ private:
 int num; //The account Id
 float balance ;
 public :
 void init(int n)
 { num=n ;
 balance=0.f ;
 }
 void consult()
 { cout <<"The account number:"<<num ;
 cout <<"\n has a balance of : "<<balance ;  } };
```

Initialization and Constructor (3/7)

1st solution:

```
account b;
b.init(100);
```

Correct but we did not do **any initialization**, in fact, the assignment is done after the declaration. In addition, we can have accounts that are manipulated without having an account number because we can declare account objects without having to call init.

Initialization and Constructor (4/7)

2nd solution:

```
account b={100,0.f} ;
```

Correct but number and balance are private so we can't access them. In addition, even if in a class all its attributes are public, they must all be initialized and in the order of their appearance in the class, which is not easy.

Initialization and Constructor (5/7)

To initialize an object, we need a new mechanism: **Constructor**.

- A constructor is a **special member function** whose job is to initialize the object; it has the same name as the class, it has no result, and it can have 0 or more parameters.

- It is called when the object is declared in an implicit way.

Initialization and Constructor (6/7)

```
class Cpoint
{ float x;
 float y;

 public:
 Cpoint (float x =0.f, float y =0.f)
 { this->x = x;
 this->y = y;
 }
};
int main()
{ Cpoint p1(5,3); //p1 has as coordinates (5,3)
 Cpoint p2(15); //p2 has as coordinates (15,0)
 Cpoint p3; //p3 has as coordinates (0,0) }
```

Initialization and Constructor (Rules) (7/7)

A constructor must adhere to the following rules:

- A constructor must have the same name as its class.
- A constructor cannot be defined with a return value not even void.
- A constructor without parameters is a default constructor.
- A constructor whose all parameters have default parameters is a default constructor.
- A constructor must be public.

Overloading constructor names

A class can have as many constructors as needed.

```
class account
{ int num;
 float balance ;
 public :
 account (int n) ;
 account(int n,float bal) ;
 account() ;
 void consult() ;
 void debit(float montant) ;
 void credit(float montant) ;
};
int main()
{ account a1 (1000); // num=1000 and balance=0
 account a2 (1500, 200); //num =1500 and balance=200
}
```

Constructor and object array

If we declare an array of objects, the default constructor will be called as many times as the size of the array.

```
account C[5]; //the constructor will be called 5 times
account *pc=new account[10]; //the constructor will be called 10
                              times
```

Constructor and object array (Exercise)

What will be the results provided by the execution of this program:

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
 public :
 point ()
 { x=0 ; y=0 ;
 cout << "** constructor 0
argument\n" ;
 }
 point (int abs)
 { x=abs ; y=0 ;
 cout << "** constructor 1
argument\n" ; }
```

```
 point (int abs, int ord)
 { x=abs ; y=ord ;
 cout << "**
constructor 2
arguments\n" ;
 }
point (point & p)
 { x=p.x ; y=p.y ;
 cout << "**copy
constructor \n" ;
 }
 void display ()
 { cout << "point : " <<
x << " " << y <<
"\n" ; } } ;
```

```
int main()
{ point a(10,20) ;
 point b(30,40) ;
 point c(10);
 point arr[3];
 for (int i=0 ; i<3 ; i++)
arr[i].display() ;
 point arr1[6]={a,b,c};
 for (int i=0 ; i<6 ; i++)
arr1[i].display() ;
 return 0;
}
```

Copy constructor (1/7)

Let's take our Point class again. It seems that it could be interesting to make a constructor from a point! This is the **copy constructor**: it is a constructor having a single object parameter of the same class. This constructor initializes an object from the content of the parameter's attributes

Copy constructor (2/7)

```
class Point
{ int x;
 int y;
 public :
 Point()
 { x = -1; Y = -1; }
 Point(int a, int b)
 { x = a; y = b; }
//Copy constructor
 Point(const Point &pt)
 { x = pt.x; y = pt.y;
 } };
...
Point pt(1,2);
Point pt2(pt); //Construction by a copy of Point
```

Copy constructor (3/7)

- C++ requires a pass by reference in the case of a copy constructor. This comes from the fact that we really have to use the object itself, not a copy.

- In the case of an object that has a pointer, when we want to copy an object, we don't copy what is pointed to, but only the address of the pointer. As a result, we end up with two different objects, but which have data that points to the same thing!

Copy constructor (4/7)

```
#include <string.h>
class PointN
{ public:
 PointN(int a, int b, char
*s="")
 { x = a;
 y = b;
 label=new char[strlen(s)+1];
 strcpy(label, s);
 }

 int x, y;
 char *label;
};
```

```
#include <iostream>
#include "PointN.cpp"
using namespace std ;
int main()
{
 PointN *a= new PointN(1,1);
 PointN *b=a; //a and b share the
same values!
 b->label="Hello";
 a->label="Mohamed";
 cout <<"The string of a is
changed but the string of b is
also worth:"<<b->label<<endl;
}
```

Copy constructor (5/7)

- We notice that we made the assignment before changing the label attribute of a and yet b->label has also changed!

-However, if we try to duplicate a in order to **separately** manage the "equal" objects pointed to by a and b:

Copy constructor (6/7)

```
#include <string.h>
class PointN

{ public:
 PointN(int a, int b, char
*s="")
 { x=a; y=b;
 label=new char[strlen(s)+1];
 strcpy(label, s);
 }
```

```
PointN(const PointN &p) //
Constructor from existing object!
 { x=p.x;
 y=p.y;
 label=new
char[strlen(p.label)+1];
 strcpy(label, p.label);
 }
 int x,y;
 char *label;
};
```

Copy constructor (7/7)

```
#include <iostream>
#include "PointN.cpp"
using namespace std ;
int main()
{ PointN *a= new PointN(1,1);
 PointN *b=new PointN(*a);
 b->label="Hello";
 a->label="Mohamed";
 cout<<" The string of a is :"<<a->label<<" and the
string of b is :"<<b->label <<endl;
}
```

Construction of member objects (1/3)

- Reminder: we speak of Member objects: when an attribute of the class is itself of type another class.

- Initializing an object of the class then requires initializing its member objects. If the member objects only have default constructors, the problem does not arise!

Otherwise, the syntax is:

Construction of member objects (2/3)

```
ClassName(parameters):member1(parameters),
member2(parameters),…
{
Body of the constructor of ClassName
}
```

Construction of member objects (3/3)

```
class Point
{ public:
 Point(int px,int py)
 { x=px; y=py; }
 private:
 int x;
 int y;
};
class Segment
{ Point origin; //Member object
 Point end; //Member object
 int thickness;
 public:
 Segment(int ox,int oy,int ex,int ey,int th): origin(ox,oy),end(ex,ey)
 {thickness=th; }
};
```

Destructors(1/5)

- Just as there is a constructor, a destructor can be specified. The latter is called when the object is destroyed, explicitly or not.

- The identifier of a constructor is that of the class preceded by the character ~ :
  - It is a method without return type;
  - It is a method without parameters, so it cannot be overloaded;
  - It is a method with public access.

Destructors(2/5)

```
class Point
 { double x, y ;
 int norm ;
 public :
 // The constructors
 Point(double xx, double yy);
 Point(int n);
 // the dstructor
 ~Point();
};
```

Thanks to function name overloading, there can be multiple constructors, but there can only be one destructor.

Destructors(3/5)

The destructor of a class C is called implicitly:
- When each object of class C disappears.
- At the end of main() for all static variables, local to main() and global variables.
- At the end of the block in which the automatic variable C is declared.
- At the end of a function with a class C argument.
- When an instance of class C is destroyed by DELETE.

Destructors(4/5)

- When an object that contains an attribute of type class C is destroyed.
- When an object of a class derived from C is destroyed.

Destructors(5/5)

```
#include <iostream>
using namespace std ;
class test
{ int attr;
 public:
 test()
 { cout<<"Default constructor"<<endl; }
 ~test()
 { cout<<"The destructor ! "<<endl; } };
void Localbloc()
{ cout<<"LOCAL BLOC : "<<endl; test CTlocal; }
```

```
int main()
{ cout<<"MAIN BLOC : "<<endl;
 test *CTmain = new test;
 cout<<"Appeal of the local bloc in the main bloc : "<<endl;
 Localbloc ();
 delete CTmain;  }
```