Chapter 7. Object-Oriented Programming (OOP) in C++

1- INTRODUCTION

Object-Oriented Programming (OOP)

OOP is a programming style based on the concept of objects. An object groups together data (information) and methods (actions that manipulate that data).

What we expect from a good program:

1. Correctness: It must do what is asked of it, according to specifications.
2. Robustness: It must respond well even in the event of incorrect or unexpected use.
3. Extensibility: It must be easy to improve or adapt.
4. Reusability: It must allow certain parts (modules) to be reused.
5. Portability: It must be able to run on different systems.
6. Efficiency: It must be fast and use little memory.

Before OOP:

Languages like C or PASCAL used structured programming, which combines algorithms (the steps to follow) and data structures (the information used).

With OOP, we organize differently to make programs more powerful and easier to manage.

Programs = algorithms + data structures

C++ is an object-oriented language. It uses classes, which group data and methods to manipulate that data.

Object-oriented programming (OOP) simplifies software creation by making it more modular and reusable.
C++ includes a library of classes that developers can use to create their programs.

In summary, OOP is based on the concept of objects, which combine data and associated actions (methods).

Methods + Data = Object

2- CONCEPT OF OBJECT, CLASS, AND ENCAPSULATION

An object is a structure that combines two essential elements:

1. Fields: These are variables that contain the object's data (e.g., a number, a text, or even another object).

2. Methods: These are functions that manipulate the object's data.

An object = data (fields) + processing (methods).

To use an object's method, we write:

objet.methode()

Example 1 :useful methods of the string type

The size() Method
The size() method allows you to find the length of the string currently stored in the string object.
This method takes no parameters and returns the length of the string. You will need to call the method as follows:

Code : C++

```
#include <iostream>
#include <string> // Nécessaire pour utiliser le type string

using namespace std;

int main() {
    string maChaine("Mohamed !");
    cout << "Longueur de la chaine : " << maChaine.size() << endl;


       system("PAUSE");

    return 0;
}
```

Result:

This program will

display:Longueur de la

chaine : 9

The size() method returns the exact length of

the string, counting all characters,

including spaces and symbols.

In the example:

La chaîne "Mohamed !" contient 9 caractères :

7 lettres : M, o, h, a, m, e, d
1 espace : après "Mohamed"
1 point d'exclamation : !

Example: This code defines a Circle class in C++, which models a 2D circle with its properties (radius and coordinates) and its methods (calculation of the perimeter and the surface area).

```
#include <iostream>
using namespace std;

class Cercle {
private:
    // Champs de type réel (float ou double selon précision)
    double rayon;
    double abs;
    double ord;

public:
    // Constructeur pour initialiser les valeurs
    Cercle(double r, double a, double o) : rayon(r), abs(a), ord(o) {}

    // Méthode pour calculer le périmètre
    double perimetre() {
        return 2 * 3.14 * rayon;
    }

    // Méthode pour calculer la surface
    double surface() {
        return 3.14 * rayon * rayon;
    }

    // Méthode pour afficher les informations du cercle
    void afficher() {
        cout << "Rayon: " << rayon << ", Abscisse: " << abs << ", Ordonnée: " << ord
<< endl;
        cout << "Périmètre: " << perimetre() << endl;
        cout << "Surface: " << surface() << endl;
    }
```

```
};

int main() {
    // Création d'un objet de type Cercle avec un rayon de 5, et des coordonnées (3,
4)
    Cercle cercle(5.0, 3.0, 4.0);

    // Affichage des informations du cercle
```

```
    cercle.afficher();

       system("PAUSE");

    return 0;
}
```

2.2 Class concept

A class is a model or blueprint that defines the characteristics and behaviors of an object. The object is an instance of this class, that is to say, a concrete example that exists in memory.

For example, a Book class can group information such as the title, author, year of publication, and number of pages. It can also include a method like getInformation() that returns these details in text form.

In C++, the declaration of a class resembles that of a structure. The main difference is that in a class, we use the keywords class and struct to declare public and private members, respectively.

A class is composed of two main parts:

1. The attributes (or fields): These are the data that describe the state of the object.
2. The methods: These are the functions that define the actions that can be performed on the object.

When an object is created, a special method called the constructor is used to initialize it. When the object is destroyed, another method, the destructor, is called to clean up the memory. The user can define their own constructors and destructors if necessary.

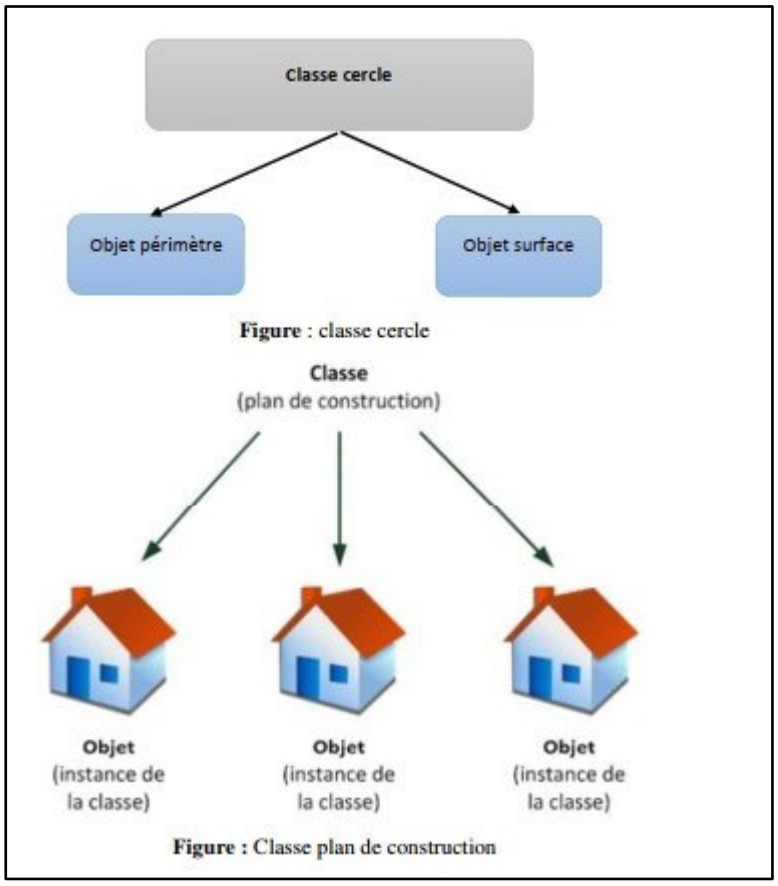


Figure : classe cercle

Figure : Classe plan de construction

Une classe est un module pour fabriquer des objets. Elle définit les méthodes, et elle décrit la structure des données.
Une classe est constituée :
De variables, ici appelées attributs (on parle aussi de variables membres) De fonctions, ici appelées méthodes (on parle aussi de fonctions membres)

Example: C++ Class Example
This example shows how to create a C++ class with attributes, a constructor, and a method to manipulate objects of that class.

```
#include <iostream>
#include <string>
using namespace std;

// Déclaration de la classe Livre
class Livre {
public:
    // Attributs de la classe
    string titre;
    string auteur;
    int anneeParution;
    int nbPages;

    // Constructeur de la classe
```

```
    Livre(string t, string a, int annee, int pages) {
        titre = t;
        auteur = a;
        anneeParution = annee;
        nbPages = pages;
    }

    // Méthode pour obtenir les informations du livre
    void obtenirInformation() {
        cout << "Titre: " << titre << endl;
        cout << "Auteur: " << auteur << endl;
        cout << "Année de parution: " << anneeParution << endl;
        cout << "Nombre de pages: " << nbPages << endl;
    }
};

int main() {
    // Création d'un objet Livre
    Livre livre1("1984", "George Orwell", 1949, 328);

    // Appel de la méthode pour afficher les informations
    livre1.obtenirInformation();

       system("PAUSE");
    return 0;
}
```

2- FUNDAMENTALS OF OOP

Object-oriented programming (OOP) is based on three fundamental principles:

- **o Inheritance (Héritage)**
- **o Encapsulation (Encapsulation)**
- **o Polymorphism(Polymorphisme**

3-1. Concept of Inheritance:

Inheritance allows you to create a new class from an already existing class. The derived (or child) class inherits the attributes and methods of the base (or mother) class. This mechanism allows for the specialization and extension of a class's functionalities without starting from scratch. In other words, the child class benefits from the properties of the parent class while adding its own specificities.

Advantages of inheritance:

• Code reuse: it is not necessary to redefine attributes and methods already present in the base class.
• Class hierarchy: it is easy to create more specialized classes from more general classes.
• Flexibility: a derived class can redefine inherited methods to adapt them to its

needs (overloading).

Example: An Animal class could have methods like seDeplacer(). A derived class like Dog could redefine this method to specify a particular behavior of the dog.

In summary, inheritance facilitates the creation of new classes based on already existing ones, thereby allowing for better organization and reuse of code.

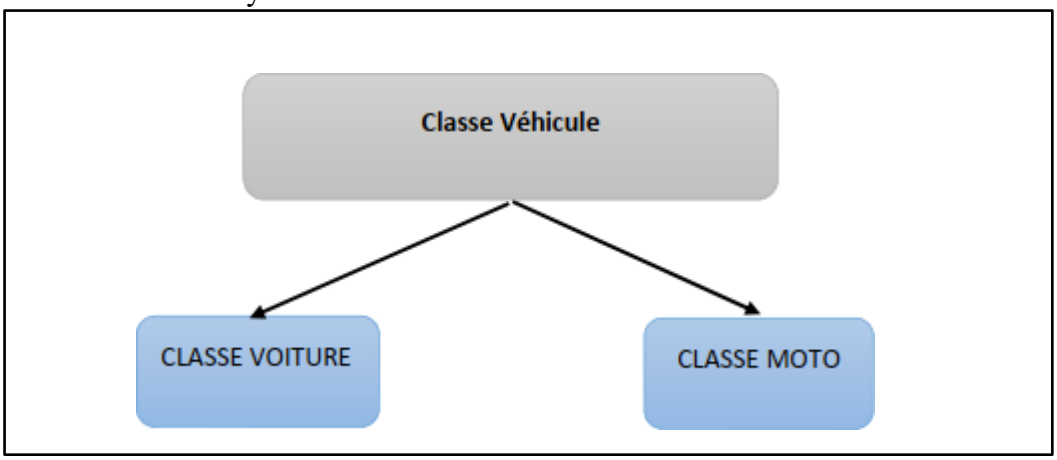
```
#include <iostream>
using namespace std;

// Classe de base
class Animal {
public:
    void manger() {
        cout << "Cet animal mange.\n";
    }
};

// Classe dérivée
class Chien : public Animal {
public:
    void aboyer() {
        cout << "Le chien aboie.\n";
    }
};

int main() {
    Chien monChien;
    monChien.manger(); // Hérité de la classe Animal
    monChien.aboyer(); // Méthode propre à la classe Chien
     system("PAUSE");
    return 0;
}
```

Example :
The car and motorcycle classes are derived from the vehicle class.



The Car and Motorcycle classes inherit the properties (fields) and methods of the Vehicle class.
Vehicle class
Fields: type, power, color; Methods: drive ( ); park ( ); Car class
Fields: steering wheel, speed; Methods: reverse ( ); park ( ); Motorcycle class
Fields: handlebars, chain; Methods: sneak ( );

Notes on the example:

The Car class:

- Retains the Vehicle Type, Color, and Power fields
- Has the new Steering Wheel and Speed fields
- Retains the Vehicle Drive() method.
- Overrides the Vehicle Park method
- Implements the Back() method. The Motorcycle class:
- Retains the Vehicle Type, Color, and Power properties
- Has the new Handlebar field

Retains the roll() method of vehicle.
- Retains the park() method of vehicle.
- Implements the sneak() method
This way, we create a hierarchy of increasingly specialized classes. The major advantage of this is that we don't have to start from scratch when we want to specialize an existing class. This way, we can develop class libraries that constitute a base that can be specialized and reusable.

A- CLASS HIERARCHY

It is possible to represent the relationship between classes in the form of a hierarchy or a tree structure. This structure begins with a parent class, called a superclass, and

develops with derived classes that become increasingly specialized. The relationship between a child class and its parent class can often be expressed by the phrase "is a" (e.g., "A dog is an animal").

Example :

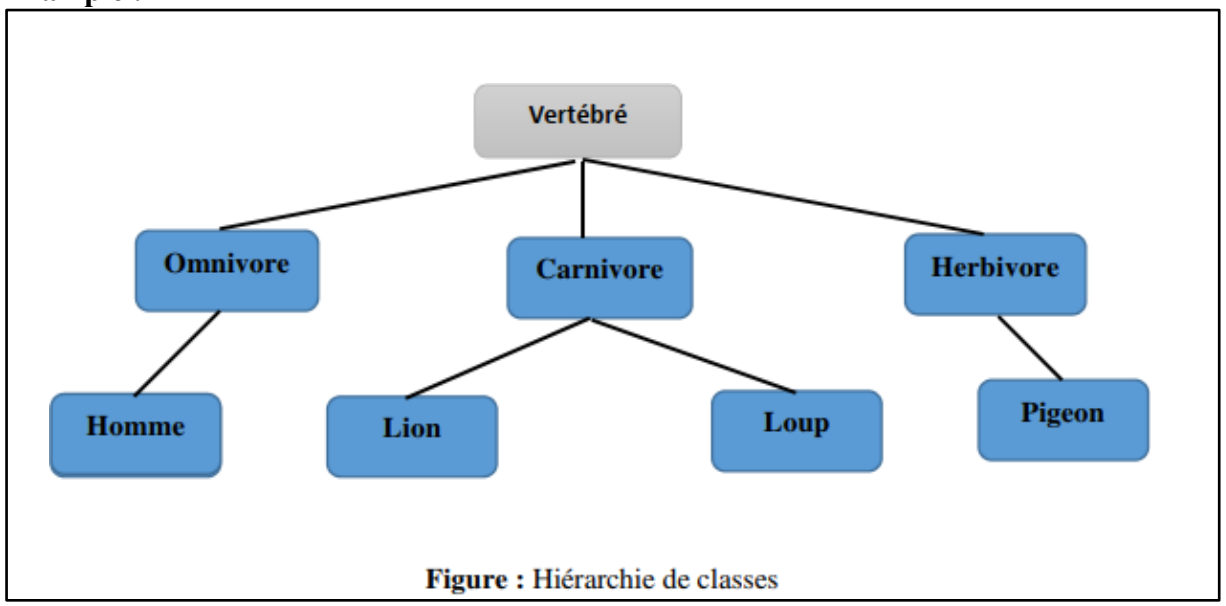


Figure : Hiérarchie de classes

Humans are said to be vertebrate omnivores. Lions are vertebrate carnivores.
Pigeons are vertebrate herbivores.

B-MULTIPLE HIERARCHY

Some object-oriented languages, such as C++, allow for multiple inheritance, which means they offer the possibility of having a class inherit from two superclasses. Thus, this technique allows for the grouping of attributes and methods from multiple classes within a single class.

3.2 CONCEPT OF ENCAPSULATION

Encapsulation involves protecting an object's data by prohibiting direct access to it. To interact with this data, it is necessary to go thru specific methods, which serve as an interface between the outside world and the object. This approach ensures that the internal details of the object are hidden, thus providing data abstraction.

Encapsulation has the advantage of making an object easier to maintain. If the internal structure of the object changes, it does not affect the other parts of the program that use this object, as they only interact with its public methods. This prevents internal modifications from disrupting the entire system, which was a problem in structured programming.

In summary, encapsulation protects data and simplifies code management by hiding implementation details, while only exposing functionalities thru methods.

In the same way, data encapsulation greatly facilitates the reuse of an object.

Encapsulation is a principle that involves grouping data and the methods that manipulate it within the same entity, called an object. Rather than separating data and functions, encapsulation unites them, making data management more consistent and secure.

Encapsulation allows for the hiding of certain internal details of the object from other parts of the program, while exposing certain methods accessible from the outside. This ensures data integrity, as they can only be modified thru authorized methods.

There are three levels of visibility in encapsulation:

1. Public: The data and methods are accessible from anywhere in the program.
2. Private: Only the methods of the class can access this data.
3. Protected: The data is accessible by the methods of the class and its derived classes.

Encapsulation helps structure the code by controlling access to data and protecting the object from unauthorized modifications.

Example

```
#include <iostream>
using namespace std;

class CompteBancaire {
private:
    double solde;  // Attribut privé, inaccessible directement
depuis l'extérieur.

public:
    // Constructeur pour initialiser le solde
    CompteBancaire(double montantInitial) {
        solde = montantInitial;
    }

    // Accesseur pour consulter le solde
    double getSolde() const {
        return solde;
```

```
    }

    // Méthode pour déposer de l'argent
    void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
            cout << "Montant déposé : " << montant << " euros\n";
        } else {
            cout << "Le montant à déposer doit être positif.\n";
        }
    }

    // Méthode pour retirer de l'argent
    void retirer(double montant) {
        if (montant > 0 && montant <= solde) {
            solde -= montant;
            cout << "Montant retiré : " << montant << " euros\n";
        } else {
            cout << "Retrait impossible. Solde insuffisant ou
montant invalide.\n";
        }
    }
};

int main() {
    // Création d'un compte bancaire avec un solde initial
    CompteBancaire monCompte(500.0);

    // Interagir avec le compte via les méthodes publiques
    cout << "Solde initial : " << monCompte.getSolde() << "
euros\n";
    monCompte.deposer(200.0);
    monCompte.retirer(100.0);
    cout << "Solde final : " << monCompte.getSolde() << " euros\n";

     system ("PAUSE");

    return 0;
}
```

3-3- NOTION OF POLYMORPHISM:

Polymorphism means that an object can have multiple forms. According to this principle, an object can be perceived differently depending on its composition or its relationship with other objects. An object is said to be polymorphic because it has several types: the type of its own class and those of the classes it inherits from.

For example, in the case of the "Vehicle" class, a "Honda" object that is an instance of the "Motorcycle" class will have "Motorcycle" as its main type, but it will also have the "Vehicle" type because a motorcycle inherits from the Vehicle class. Thus, "Honda" is also a vehicle.

Exemple :
Prenons l'exemple d'un programme qui calcule le périmètre et la surface (aire) d'in cercle et d'une sphère.
1- Déduire l'arbre d'héritage les deux formes géométriques ont comme point commun un nom, un périmètre, un aire et un volume. Ces caractéristiques communes apparaissent dans une classe nommée figure se trouvant à la racine de l'arbre d'héritage. Par la suite, pour déterminer le périmètre et l'aire d'un cercle, on doit connaitre son rayon, il en va de même pour calculer l'aire et le volume d'une sphère. Cette information supplémentaire particulière à chaque figure conduit à la définition de deux classes distinctes : cercle descendant de la classe figure, et sphère, la descendante de la classe cercle. La classe cercle contient comme donnée la longueur du rayon.

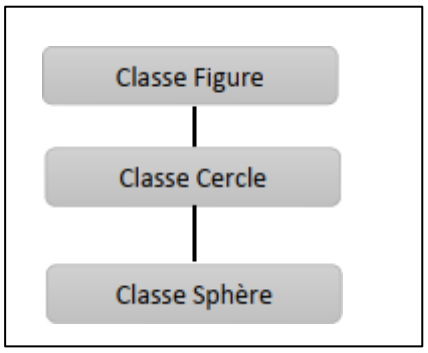


Figure: Inheritance tree of the Figure Class

1- Determine the methods of each class
The Figure class must allow for the display of the information it contains.

Class Figure Fields: nameFigure, Area, Perimeter, Volume; Methods: displayInfo(); Class Circle Fields: Radius; Methods:

obtenirRayen ( ) ;
calculerperimetre ( ) ;
calculerAire( ) ;
Classe Sphère
Champs :
Méthodes :
calculerAire( ) ;
calculerVolume ( ) ;

Remarks

The call to the calculateArea() function is identical for an object of the Circle class and for an object of the Sphere class, even tho the calculations for the area of a circle and the area of a sphere are different, this is the principle of polymorphism.

Object-oriented programming (OOP) uses common methods in languages like C++, Java, and PHP. Among these, we mainly find constructors and destructors.

4-1- The constructors

Constructors are used to create an object in memory. They initialize the attributes and bind the methods to the data, while also setting up the inheritance between the classes. Each instance of an object has its own attributes, but the methods are shared in memory. A constructor can be defined or left to the compiler's option, which then creates the object automatically. Some languages, like Java, allow for multiple constructors for the same object, enabling different initializations.

4-2- The destructors

The role of destructors is to free up the memory used by the object when it is no longer needed. They delete the object instance and clean up resources, such as memory and method code. In general, an object may not have an explicit destructor, and it is the compiler that takes care of its deletion. Like constructors, destructors can be overloaded in some languages.

In summary:

Constructor: Creates and initializes an object.
Destructor: Destroys an object and releases its resources in memory.

Example

```
#include <iostream>
using namespace std;

// Classe de base
class Animal {
public:
    // Fonction virtuelle pour permettre le polymorphisme
    virtual void parler() {
        cout << "Cet animal fait un bruit.\n";
    }
};

// Classe dérivée : Chien
class Chien : public Animal {
public:
    void parler() override { // Redéfinition de la méthode
        cout << "Le chien aboie.\n";
    }
};

// Classe dérivée : Chat
class Chat : public Animal {
public:
    void parler() override { // Redéfinition de la méthode
        cout << "Le chat miaule.\n";
    }
};

int main() {
    // Création d'objets spécifiques
    Animal* animal1 = new Chien();  // Pointeur de type Animal pointant sur
un Chien
    Animal* animal2 = new Chat();  // Pointeur de type Animal pointant sur
un Chat

    // Appel des méthodes via le polymorphisme
    animal1->parler(); // Appelle la méthode parler() redéfinie dans Chien
    animal2->parler(); // Appelle la méthode parler() redéfinie dans Chat

    // Libération de la mémoire
    delete animal1;
    delete animal2;
      system("PAUSE");
    return 0;
}
```