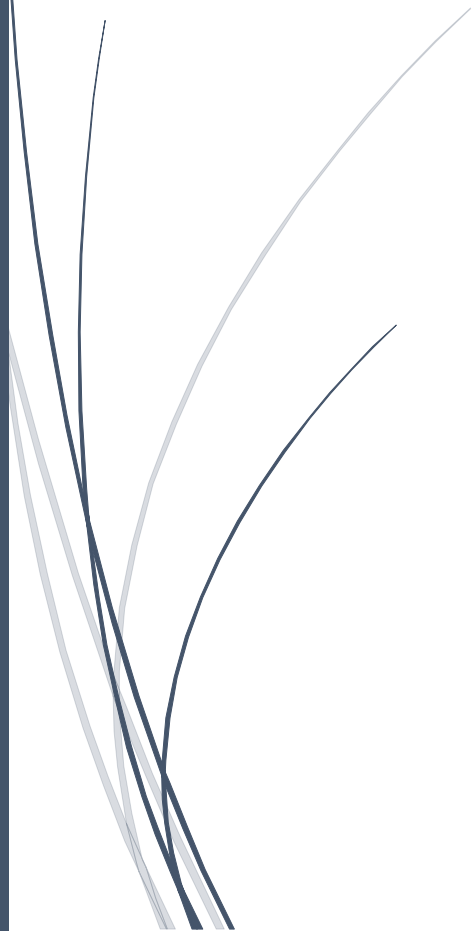
2025

Object Oriented Programming Cpp

By Javaid Iqbal Awan (BSCS)



Javaid Iqbal Awan (BSCS)
https://www.linkedin.com/in/javaid-iqbal-283163238/
drjav147@gmail.com
https://github.com/JavaidIqbal786

Table of Contents

Chapter No 01: Introduction to OOP

Object Orientation:

It is a **technique in which we visualize our programming problems** in the form of objects and their interactions as happens in real life.

Classes and Objects:

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

class Student {     //Class
    public:
    //Attribute
    int rollNo;
    //Member Function
    void setRollNo (int num){
        cout<<"Student Roll No: "<<num<<endl;
    }
};

main (){
    Student aStudent;   //Object
    aStudent.setRollNo(1);
    // OR
    Student *ptr_student = new Student ();
    ptr_student->setRollNo(5);
    return 0;
}
```

- **Objects** are **entities** in the real world. An object is something **tangible or conceptual** (e.g. Person, Car, Tree, House, Time, Date etc.). An object has **State** (attributes), **Well-defined behavior** (operations) and **Unique identity**.
- **Class** is like a **Sketch or prototype or blueprint** of these entities (objects). Object is an instantiation of a user defined type or class. Once we have defined a class we can create as many objects as we need.

Syntax:

```
Class ClassName {

};

main(){

        ClassName obj1, obj2, obj3;

}
```

Access Modifiers / Access Specifiers:

Private (Default) *data & methods accessible inside class*

Public *data & methods accessible to everyone*

Protected *data & methods accessible inside class & to its derived class*

Dot Operator (.) is use to access properties of the object via variable name. e.g. t1.name

Arrow Operator (→) is use to access properties of the object via a pointer. E.g.

Student *ptr_student = new Student ();

ptr_student->setRollNo(5);

Member Functions:

Member functions are the functions that operate on the data encapsulated in the class.

Public member functions are the interface to the class.

New Operator:

New operator can be used to create an objects at runtime. Allocates memory **dynamically** for an object or array **on the heap** and **returns a particular pointer**.

Initialize the object if constructor is defined. The constructor is called **immediately after memory allocation** when the new operator is used.

The allocated memory must eventually be **deallocated using the delete operator** to prevent memory leaks.

Chapter No 02: 4 Pillars of OOP

- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**

1. Encapsulation:

*Encapsulation is **wrapping up** of data & member functions in a single unit called class.*

OR

Encapsulation means "We have enclosed all the characteristics of an object in the object itself".

Information Hiding:

Information hiding is one the most important principals of OOP inspired from the real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner.

```
class Teacher { // class

    private:
    double salary;  // Data-Hiding

    public:     // Access modifier
    //properties(attributes)
    string name;
    string dept;
    string subject;

    // methods (member functons)
    void changeDept (string newDept) {
        dept = newDept;
    }

    void getInfo () {
        cout<<"Name: "<<name<<endl;
        cout<<"Department: "<<dept<<endl;
        cout<<"Subject: "<<subject<<endl;
        cout<<"Salary: "<<salary<<endl;

    }
    // setter
    void setSalary (double s){      // setter fuction is used to set private values
        salary = s;
    }

    // getter
    double getSalary(){     // getter function is used to get private values indirectly
        return salary;
    }
};
```

Encapsulation

Encapsulation helps us to hide sensitive data. We did it by specifying **private access modifier** to our sensitive data.

Advantages of Encapsulation:

- Simplicity and clarity
- Low complexity
- Better understanding

Setter Fuction is used to set private values and **Getter Function** is used to get that private values indirectly.

2. Abstraction:

Hiding all unnecessary details and showing only the important parts.

OR

Capture only those details about an object that are relevant to current perspective.

For Example:

Ali is a PHD student and he teaches BS students.

So here the object Ali has **two perspectives** one is his **student perspective** and the other is his **teacher perspective**.

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

class Person {
    public:
        int id, age;
        string name;
        float cgpa, salary;
        void studies(){
            cout<<"PHD Computer Science"<<endl;
        }
        void teaches(){
            cout<<"Teach BS Students"<<endl;
        }
};
class Student {

};
class Teacher {

};

main (){
    return 0;
}
```

Abstraction

Abstract Classes:

Abstract classes is like a blueprint for other classes that are going to be implemented.

- Abstract classes are used to **provide a Bass/Parent class** from which **other classes can be derived**.
- They **cannot be instantiated** (They have no objects) and are meant to be inherited.
- Abstract classes are typically used to **define an interface** for derived classes.

Advantages of Abstraction:

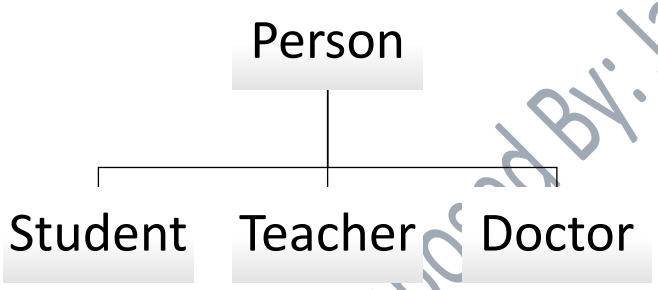
- It help us understanding and solving a problem using object oriented approach as it hides extra irrelevant details of objects.
- Focusing on single perspective of an object provides us freedom to change implementation for other aspects of an object later.

3. Inheritance:

*When properties and member functions of **Bass/Parent class** are passed on to the **Derived/Child class***. Private members are never inherited.

Inheritance is **"IS A"** or **"IS A KIND OF"** relationship.

Each derived class is a kind of its base class.




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

class Person {  // Parent Class
public:
    string name;
    int age;

    // Constructor
    Person(string name, int age) {
        this->name = name;
        this->age = age;
    }
};

class Student : public Person {  // Child Class
public:
    string rollNo;

    // Constructor for Student, calls the Person constructor
    Student(string name, int age, string rollNo) : Person(name, age) {
        this->rollNo = rollNo;
    }

    // Method to print Student info
    void getInfo() {
        cout << "Roll No: " << rollNo << endl;
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main() {
    // Create a Student object with required parameters
    Student s1("Javaid", 21, "1001");
    s1.getInfo();
    return 0;
}
```

Inheritance

Inheritance Advantages:

- Reuse
- Less redundancy
- Increase maintainability

Multiple Inheritance:

Sometimes we need to inherit characteristics from more than one class that is called multiple inheritance. We can inherit a class from many classes and can use their functions without the need to write them again.

//Programs to demonstrate simple multiple inheritance

Example: 01

```cpp
// Program to demonstrate simple Multiple inheritance
#include<iostream>
#include<stdio.h>
using namespace std;

class Women {
    public:
        void walk(){
            cout<<"Mermaid is walking"<<endl;
        }
};

class Fish {
    public:
        void swim(){
            cout<<"Mermaid is swiming"<<endl;
        }
};

class Mermaid : public Women, public Fish {

};
main (){
    Mermaid m;
    m.swim();
    m.walk();
    return 0;
}
```

Output

```
Mermaid is swiming
Mermaid is walking

--------------------------------
Process exited after 0.06054 seconds with return value 0
Press any key to continue . . .
```

Example: 02

```cpp
#include<iostream>
#include<stdlib.h>
using namespace std;

//Multiple Inheritance in case of Amphibious vehicle
class Vehicle {
    public:
        void changeGear(){
            cout<<"I am Vehicle ChangeGear() Function!"<<endl;
        }
};

class WaterVehicle: public Vehicle {
    public:
        void Float(){
            cout<<"I am Float() Function of Water Vehicle!"<<endl;
        }
};

class LandVehicle: public Vehicle {
    public:
        void Move(){
            cout<<"I am Move() Function of Land Vehicle!"<<endl;
        }
};

class AmphibiousVehicle: public WaterVehicle, public LandVehicle {

};

main (){
    AmphibiousVehicle amphVehicle;
    amphVehicle.Float();
    amphVehicle.Move();
}
```

Output

```
I am Float() Function of Water Vehicle!
I am Move() Function of Land Vehicle!

--------------------------------
Process exited after 0.05883 seconds with return value 0
Press any key to continue . . .
```

4. Polymorphism:

Poly means multiple and **morph** means forms. *Polymorphism refers to existence of different forms of a single entity.* Polymorphism is the ability of objects to take on **different forms or behave** in different ways **depending on the context** in which they are used. Polymorphism is a powerful tool to develop flexible and reusable systems.

Constructor overloading is an example of polymorphism.

Types of polymorphism:

There are two types of polymorphism

Compile time polymorphism:

(e.g. Constructor Overloading, Function Overloading, and Operator Overloading)

- **Constructor Overloading:**

  Contains two or more constructors of the same name but parameters are different.

- **Function Overloading:**

  Function overloading is when we are defining two or more functions with the same name in the same class but they only differs in term of their parameters.

Runtime polymorphism:

(e.g. Function Overriding, Virtual Functions)

- **Function Overriding:**

  Both parent and child class contain the same function with different implementation.

  The parent class function is said to be overridden.

- **Virtual Functions:**

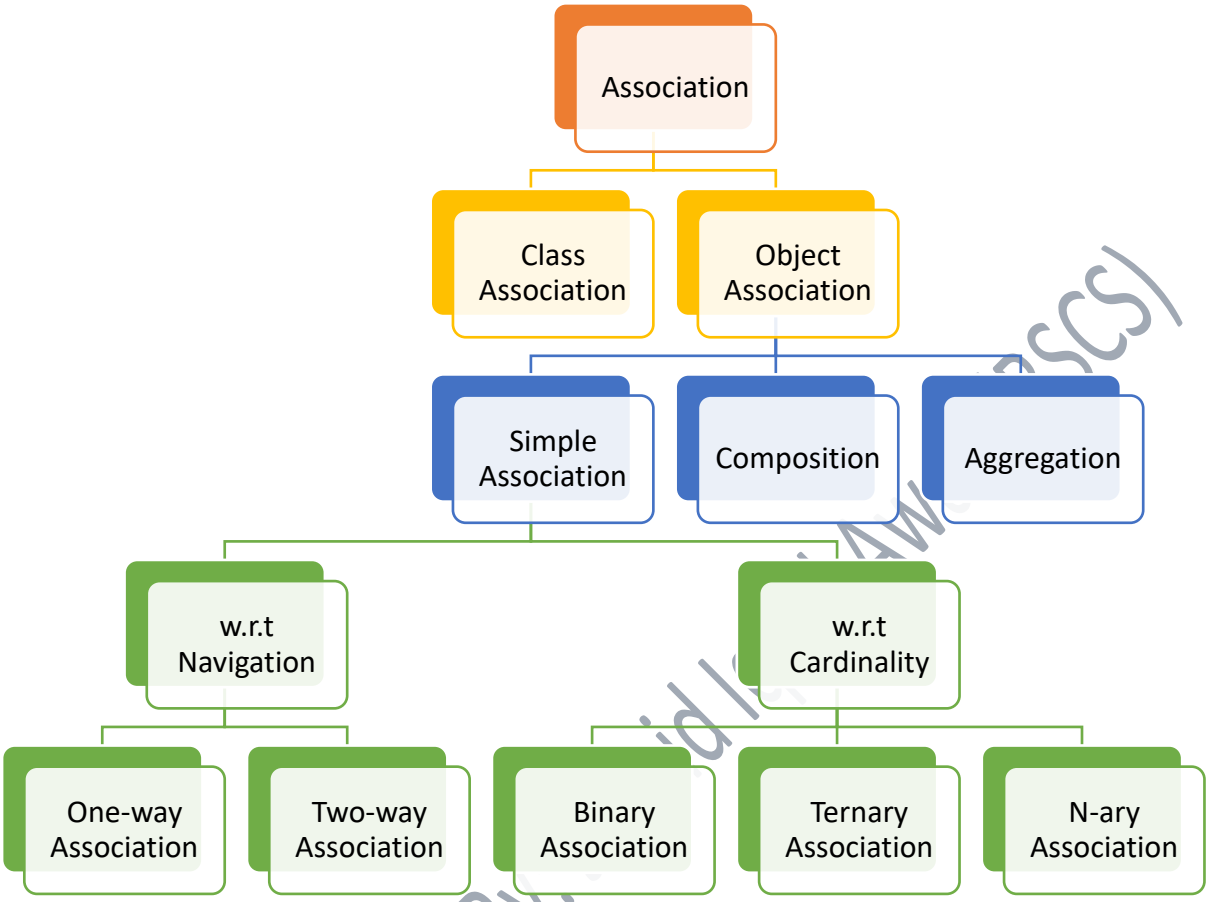
  A virtual function is a member function that you expect to be redefined in derived classes.

  - Virtual Functions are Dynamic in nature.
  - Defined by the keyword "**virtual**" inside a parent class and always declared in a parent class and overridden in a child class.
  - A virtual function is called during runtime.

Operator Overloading:

Operator overloading is the feature in C++ that allows you to define basic operators (+, - , *, / etc) to work with user-defined types (classes) in the similer way as we perform them on basics build-in types (like int, float, double etc).

Chapter No 03: Association



Association:

Interaction of different objects in object oriented model is known as association. An object keeps association with other objects to delegate tasks.

Kinds of Association:

- **Class Association**

  Class association is implemented in term of inheritance. Inheritance implements generalization/specialization relationship between objects. Inheritance is considered class association.

  - In case of public inheritance it is "IS-A" relationship.
  - In case of private inheritance it is "Implemented in term of" relationship.

- **Object Association**

  It is the interaction of stand-alone objects of one class with other objects of another class. Types of object association are:

  - Simple Association
  - Composition
  - Aggregation

1. Simple Association:
The two interacting objects have no intrinsic relationship with another object. It is the weakest link between objects. It is the reference by which one object can interact with some other object. It is generally called as "association" instead "simple association".

Kinds of Simple Association:

i. **w.r.t direction (Navigation)**
   - **One-way association:** In one-way association we navigate only in one direction. It is denoted by an arrow towards server object.

     For example:

     Ali lives in a house.

     Ali drives a car.

   - **Two-way association:** In two-way association we can navigate in both directions. It is denoted by line between the associated objects.

     For example:

     Employee works for Company

     Company hires Employees

ii. **w.r.t number of objects (Cardinality)**
   - **Binary Association:** It associates objects of exactly two classes. For example

     Employee works for Company.

     Ali drives a car.

   - **Ternary Association:** It associates objects of exactly three classes. For example

     Association of Student, Course and Teacher

   - **N-ray Association:** It is the association between more than 3 classes its practical examples are very rare.

2. Composition:
An object maybe composed of other smaller objects, the relationship between the 'part' objects and the 'whole' object is known as composition. Composition is denoted by a line with filled-diamond head towards the composer object.

Example: Composition of chair

A chair is composed of Seat, Back, Arms and Legs

Composition is a stronger relationship because composed objects become the part of the composer. Composed objects cannot exists independently.

3. Aggregation:
An object may contain collection (aggregate) of other objects, **the relationship between the container and the contained object is called aggregation.** Aggregation is represented by a line with unfilled-diamond head towards the container.

Example:

Room contains Bed, Chair, Table, and Cupboard

Aggregate object can exist independently.

Class Compatibility:

A class is behaviorally compatible with another if it supports all the operations of the other class. Such a class is called **subtype**. Derived class is usually a subtype of Base class.

Chapter No 04: Constructor & Destructor

Constructor:

*Constructor is a special method which **invoked automatically** at the time of object creation. Used for **initialization**.*

- Constructor is a special function having same name as class
- Construct does not have a return type
- Only called once (automatically), at object creation
- Constructor is always declared publicly
- Memory allocation happens which constructor is called

Constructor Overloading

Contains two or more constructors of the same name but parameters are different.

Types of Constructors:

- **Parameterized** contains parameters
- **Non-parameterized** does not contain parameters
- **Copy Constructor** Special constructor (default) used to copy properties of one object to another.

```
Teacher t1 ("Javaid", "Computer Science", "C++", 25000);
Teacher t2(t1); // Copy Constuctor
```

```
// non-parameterized Constuctor
Teacher (){
    cout<<"Hi, I am a constructor!"<<endl;
    dept = "Computer Science";
}

// parameterized Consturctor
Teacher(string n, string d, string s, double sal){
    name = n;
    dept = d;
    subject = s;
    salary = sal;
}
```

Copy Constructor:

Copy constructor are used when

- Initializing an object at the time of creation (An object is created in terms of a pre-existing object)
- When an object is passed by value to a function.

Shallow & Deep Copy:

A **shallow copy** of an object copies all of the member values from one object to another.

A **deep copy**, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the member points to.

This Pointer:

this is a special pointer in C++ that points to the current object.

Syntax:

this → prop

Destructor:

*Destructor is used to **deallocate** memory*. Destructor has the same name as the class. **~ Sign** is used before it.

Chapter No 05

Static Keyword:

Static Data Members:

Static data members can be created without any object of a class. Static data members belong to the class rather than any specific instance (object) of the class. They are shared among all the instances (objects) of the class.

Static data member is declared inside the class but it is initialized/defined outside the class.

Syntax:

class ClassName{

…

static DataType VariableName;

};

ClassName::StaticMember

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

class Employee {
    private:
    static int count;
    const string empId;

    public:
        // constructor
        Employee(string id) : empId(id) {
        count++;
    }
        // Destructor
        ~Employee(){
        count--;
    }

    static int getCount()   // Static Function
        {
            return count;
        }

    string getEmpId () const{
        return empId;
    }
};

int Employee::count = 0;

int main (){
    Employee E1("E001"), E2("E002"), E3("E003");
    const Employee E4("E004");
    cout<<"Employee 1 id: "<<E1.getEmpId()<<endl;
    cout<<"Employee 2 id: "<<E2.getEmpId()<<endl;
    cout<<"Employee 3 id: "<<E3.getEmpId()<<endl;
    cout<<"Employee 4 id: "<<E4.getEmpId()<<endl;
    cout<<"total number of employees: "<<Employee::getCount();
    return 0;
}
```

Use of Satic keyword

Static Variables:

Variables declared as static in a function are **created & initialized once** for the lifetime of the program. **// in Function**

Static Variables in a class are created & initialized once. They are **shared by all the objects** of the class. **// in Class**

Static Objects:

Static objects remains lifetime of the program.

Friend Functions:

*The functions which are **not member functions** of the class yet they **can access all private members** of the class are called **friend functions**.*

In order to access the member variables of class, we must make friend function of that class. The function **DoSomething** is used to access data members of class.

- Prototypes of friend functions appear in the class definition
- Friend functions are not member functions
- Friend functions can be placed anywhere in the class
- Access specifiers has no impact on friend functions or classes