Object Oriented Programming in C++

- Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.
- Object-oriented programming has several advantages over procedural programming
- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
- There are some basic concepts that act as the building blocks of OOPs i.e.
    - Class
    - Objects
    - Encapsulation
    - Abstraction
    - Polymorphism
    - Inheritance
    - Dynamic Binding
    - Message Passing

Class

- The building block of C++ that leads to Object-Oriented programming is a Class.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A class is like a blueprint for an object.
- For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc.
- So here, the Car is the class, and wheels, speed limits, and mileage are their properties.
- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

Object

- An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
/ C++ Program to show the syntax/working of Objects as a

// part of Object Oriented PProgramming

#include <iostream>

using namespace std;


class person {

   char name[20];

   int id;


public:

   void getdetails() {}

};


int main()

{


   person p1; // p1 is a object

   return 0;

}
```

- Objects take up space in memory and have an associated address
- When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

Encapsulation

- In normal terms, Encapsulation is defined as wrapping up data and information under a single unit.

- In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.
- Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,
- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.
- Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.
- In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data
- This is what **Encapsulation** is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".
- **Two Important property of Encapsulation**

1. **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.
2. **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

Features of Encapsulation

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Encapsulation improves readability, maintainability, and security by grouping data and methods together.
5. It helps to control the modification of our data members.

Encapsulation also leads to data abstraction.

Role of Access Specifiers in Encapsulation

- Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members. There are three types of access specifiers in C++:
- **Private:** Private access specifier means that the member function or data member can only be accessed by other member functions of the same class.
- **Protected: A protected** access specifier means that the member function or data member can be accessed by other member functions of the same class or by derived classes.
- **Public:** Public access specifier means that the member function or data member can be accessed by any code.
- By **default**, all data members and member functions of a class are made **private** by the compiler.
- The process of implementing encapsulation can be sub-divided into two steps:

1. Creating a class to encapsulate all the data and methods into a single unit.
2. Hiding relevant data using access specifiers.

Abstraction in C++

- Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

Types of Abstraction

1. **Data abstraction –** This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction –** This type only shows the required information about the implementation and hides unnecessary information.

Abstraction using Classes

- We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files**
- One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

- **Abstraction using Access Specifiers**
- Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:
- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

C++ Polymorphism

- The word "polymorphism" means having many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.
- So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**

Compile-Time Polymorphism

- This type of polymorphism is achieved by function overloading or operator overloading.
- **Function Overloading**
- When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading.
- Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**.
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.
- There are certain Rules of Function Overloading that should be followed while overloading a function.
- **Operator Overloading**
- C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can make use of the addition operator (+) for string class to concatenate two strings.

- We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Runtime Polymorphism

- This type of polymorphism is achieved by **Function Overriding**.
- Late binding and dynamic polymorphism are other names for runtime polymorphism.
- The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

Virtual Function

- A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.
- **Some Key Points About Virtual Functions:**
- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime
- What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.
- Pure Virtual Functions
- It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

```
class Shape {
  protected:
    int width, height;
  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
```

```
        virtual int area() = 0;

    };
```

Constructors in C++

- **Constructor in C++** is a special method that is invoked automatically at the time of object creation.
- It is used to initialize the data members of new objects generally.
- The constructor in C++ has the same name as the class or structure.
- It constructs the values i.e. provides data for the object which is why it is known as a constructor.
- Constructor is a member function of a class, whose name is the same as the class name.
- Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically when an object of the same class is created.
- Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as a constructor.
- Constructors do not return value, hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- A constructor cannot be declared virtual.
- **Syntax for Defining the Constructor Within the Class**

```
<class-name> (list-of-parameters)

{

   // constructor definition

}
```

- **Syntax for Defining the Constructor Outside the Class**

```
<class-name>: :<class-name>(list-of-parameters)

{

   // constructor definition

}
```

Characteristics of Constructors in C++

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though they can be declared in the private section of the class.

- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- A constructor can not be declared virtual.
- A constructor cannot be inherited.
- The addresses of the Constructor cannot be referred to.
- The constructor makes implicit calls to **new** and **delete** operators during memory allocation.
- Private constructors are typically used when the class is not intended to be used as a standalone object but rather as a base class for inheritance. In this case, the derived classes can call the private constructor of the base class using the base keyword to create instances of the base class. This allows the derived classes to control how and when the base class is instantiated and can be used to implement the Singleton pattern, where only one instance of a class is allowed to be created.

Types of Constructor in C++

- Constructors can be classified based on in which situations they are being used. There are 4 types of constructors in C++:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Move Constructor**

- **Default Constructor in C++**
- A constructor without any arguments or with the default value for every argument is said to be the **Default constructor**.
- A constructor that has zero parameter list or in other sense, a constructor that accept no arguments is called a zero argument constructor or default constructor.
- If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation.
- If the default constructor is defined explicitly in the program by the programmer, then the compiler will not defined the constructor implicitly, but it calls the constructor implicitly.

Parameterized Constructor in C++

- Parameterized Constructors make it possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

- When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly create the default constructor and hence create a simple object as: Student s; will flash an error
- ***Important Note:*** *Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.*
- **Uses of Parameterized Constructor**
- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.
- **Default Arguments with C++ Parameterized Constructor**
- Just like normal functions, we can also define default values for the arguments of parameterized constructors. All the rules of the default arguments will be applied to these parameters.

Copy Constructor in C++

- A copy constructor is a member function that initializes an object using another object of the same class.
- Copy constructor takes a reference to an object of the same class as an argument.

```
ClassName (ClassName &obj)

{

 // body_containing_logic

}
```

- A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.
- Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- Copy constructor takes a reference to an object of the same class as an argument.
- The process of initializing members of an object through a copy constructor is known as copy initialization.
- It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.
- The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.
- The copy constructor is used to −

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Dynamic initialization of object in C++

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- It can be achieved by using constructors and by passing parameters to the constructors.
- This comes in really handy when there are multiple constructors of the same class with different inputs.
- The constructor used for allocating the memory at runtime is known as the dynamic constructor.
- The memory is allocated at runtime using a new operator and similarly, memory is deallocated at runtime using the delete operator.

```
class streg {
   const char* p;

public:
   // default constructor
   streg()
   {

      // allocating memory at run time
      p = new char[6];
      p = “Hello";
   }

   void display() { cout << p << endl; }
};

int main()
{
   streg obj;
```

```
    obj.display();
}
```

Destructors in C++

- A destructor is also a special member function as a constructor.
- Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope.
- Destructors release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of object creation
- The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

Properties of Destructor

- The destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of the destructor.

When is the destructor called?

- A destructor function is called automatically when the object goes out of scope:
- The function ends
- The program ends
- A block containing local variables ends
- A delete operator is called
- ***Note: destructor*** *can also be called explicitly for an object.*
- We can call the destructors explicitly using the following statement:

  object_name.~class_name()

  How are destructors different from normal member functions?
- Destructors have the same name as the class preceded by a tilde (~)
- Destructors don’t take any argument and don’t return anything

When do we need to write a user-defined destructor?

- If we do not write our own destructor in class, the compiler creates a default destructor for us.

- The default destructor works fine unless we have dynamically allocated memory or pointer in class.
- When a class contains a pointer to memory allocated in the class, we should write a destructor to release memory before the class instance is destroyed.
- This must be done to avoid memory leaks.

Can a destructor be virtual?

- Yes, In fact, it is always a good idea to make destructors virtual in the base class when we have a virtual function.

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Namespace

- Namespace provide the space where we can define or declare identifier i.e. variable, method, classes.

- Using namespace, you can define the space or context in which identifiers are defined i.e. variable, method, classes. In essence, a namespace defines a scope.

- **Advantage of Namespace to avoid name collision.**

- Example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

- A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries.

- The best example of namespace scope is the C++ standard library (std) where all the classes, methods and templates are declared. Hence while writing a C++ program we usually include the directive using namespace std;

Defining a Namespace:

- A namespace definition begins with the keyword namespace followed by the namespace name as follows

```
namespace  namespace_name

{

   // code declarations i.e. variable  (int a;)

   method (void add();)

   classes ( class student{};)

}
```

- It is to be noted that, there is no semicolon (;) after the closing brace.

- To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:
- namespace_name: :code; // code could be variable , function or class.

The using directive

```
#include <iostream>

using namespace std;

// first name space

namespace first_space

{

  void func()

  {

     cout << "Inside first_space" << endl;

  }

}

 // second name space

namespace second_space

{

  void func()

  {

     cout << "Inside second_space" << endl;

  }

}

using namespace first_space;

int main ()

{

   // This calls function from first name space.

  func();

  return 0;

}
```