C++ OOP Basics

What is OOP?

OOP stands for Object-Oriented Programming. An object-oriented programming language uses objects in its programming. Programming with object-oriented concepts aims to emulate real-world concepts such as inheritance, polymorphism, abstraction, etc., in a program.

C++ language was designed with the main intention of adding object-oriented programming to C language. As the size of the program increases, the readability, maintainability, and bug-free nature of the program decrease. 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.

This was the major problem with languages like C, which relied upon functions or procedures (hence the name procedural programming language). As a result, the possibility of not addressing the problem adequately was high. Also, data was almost neglected, and data security was easily compromised. Using classes solves this problem by modeling the program as a real-world scenario.

Difference between Procedure Oriented Programming and Object-Oriented Programming

Procedure Oriented Programming

- Consists of writing a set of instructions for the computer to follow
- The main focus is on functions and not on the flow of data
- Functions can either use local or global data
- Data moves openly from function to function

Object-Oriented Programming

- Works on the concept of classes and objects
- A class is a template to create objects
- Treats data as a critical element
- Decomposes the problem into objects and builds data and functions around the objects

Basically, procedural programming involves writing procedures or functions that manipulate data, while object-oriented programming involves creating objects that contain both data and functions.

Basic Elements in Object-Oriented Programming

- **Classes** - Basic template for creating objects. This is the building block of object-oriented programming.
- **Objects** – Basic run-time entities and instances of a class.
- **Data Abstraction & Encapsulation** – Wrapping data and functions into a single unit.
- **Inheritance** – Properties of one class can be inherited into others.
- **Polymorphism** – Ability to take more than one form.
- **Dynamic Binding** – Code which will execute is not known until the program runs.
- **Message Passing** – Message (information) call format.

Benefits of Object-Oriented Programming

Object-oriented programming has many advantages. Listed below are a few:

- Programs involving OOP are faster and easier to execute.
- By using objects and inheritance, it provides a clear structure for programs and improves code reusability.
- It makes the code easier to maintain, modify, and debug.
- Principle of data hiding helps build secure systems.
- Multiple objects can co-exist without any interference.
- Software complexity can be easily managed so that even the creation of fully reusable software with less code and shorter development time is possible.

C++ Classes & Objects

1. Classes

- **Similarities with Structures**: Both can group data.
- **Differences from Structures**:
    - In **structures**, all members are **public** by default → no data hiding.
    - Structures cannot contain **functions**.
    - Hence, **structures lack data security and abstraction**.
- **Definition**:
    - Classes are **user-defined data types**.

    - They act as a **template** for creating objects.
    - They can have:
        - **Variables** (data members)
        - **Functions** (member functions)
- **Syntax**:

```
class class_name {
  // body of the class
};
```

---

2. Objects

- Objects are **instances of a class**.
- They can **access class attributes and methods**.
- Permissions are controlled using **access modifiers** (public, private, protected).
- **Syntax**:

```
class class_name {
  // body of the class
};


int main() {
  class_name object_name; // object creation
}
```

Class Attributes & Methods

Class attributes and methods are variables and functions that are defined inside the class. They are also known as class members altogether.

Consider an example below to understand what class attributes are

```
#include <iostream>

using namespace std;
```

```
class Employee
{
   int eID;
   string eName;
   public:
};

int main()
{
   Employee Harry;
}
```

A class named Employee is built and two members, eID and eName are defined inside the class. These two members are variables and are known as class attributes. Now, an object named Harry is defined in the main. Harry can access these attributes using the dot operator. But they are not accessible to Harry unless they are made public.

```
class Employee
{
public:
   int eID;
   string eName;
};

int main()
{
   Employee Harry;
   Harry.eID = 5;
   Harry.eName = "Harry";
   cout << "Employee having ID " << Harry.eID << " is " << Harry.eName << endl;
}
```

Output

```
Employee having ID 5 is Harry
```

Class methods are nothing but functions that are defined in a class or belong to a class. Methods belonging to a class are accessed by their objects in the same way that they access attributes. Functions can be defined in two ways so that they belong to a class.

Defining inside the class

An example that demonstrates defining functions inside classes is:

```
class Employee
{
public:
  int eID;
  string eName;

  void printName()
  {
    cout << eName << endl;
  }
};
```

Defining outside the class

Although a function can be defined outside the class, it needs to be declared inside. Later, we can use the scope resolution operator (::) to define the function outside.

An example that demonstrates defining functions outside classes is:

```
class Employee
{
public:
  int eID;
  string eName;
  void printName();
};

void Employee::printName()
{
  cout << eName << endl;
}
```

Objects Memory Allocation

Objects Memory Allocation in C++

The way memory is allocated to variables and functions of the class is different even though they both are from the same class. The memory is only allocated to the variables of the class when the object is created.

The memory is not allocated to the variables when the class is declared. At the same time, single variables can have different values for different objects, so every object has an individual copy of all the variables of the class. But the memory is allocated to the function only once when the class is declared. So the objects don’t have individual copies of functions; only one copy is shared among each object.

Static Data Members in C++

When a static data member is created, there is only a single copy of the data member which is shared between all the objects of the class. Usually, every object has an individual copy of the attributes unless specified statically.

Static members are not defined by any object of the class. They are exclusively defined outside any function using the scope resolution operator.

An example of how static variables are defined is:

```
class Employee
{
public:
  static int count; // returns number of employees
  string eName;

  void setName(string name)
  {
    eName = name;
    count++;
  }
};

int Employee::count = 0; // defining the value of count
```

Static Methods in C++

When a static method is created, they become independent of any object and class. Static methods can only access static data members and static methods. Static methods can only be accessed using the scope resolution operator. An example of how static methods are used in a program is shown.

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

class Employee
{
public:
  static int count; // static variable
  string eName;

  void setName(string name)
  {
    eName = name;
    count++;
  }

  static int getCount() // static method
  {
    return count;
    }
};

int Employee::count = 0; // defining the value of count

int main()
{
    Employee Harry;
    Harry.setName("Harry");
    cout << Employee::getCount() << endl;
}
```

Output:

```
1
```

Friend Functions & Classes

Friend functions are those functions that have the right to access the private data of members of the class even though they are not defined inside the class. It is necessary to write the prototype of the friend function.

Declaring a friend function inside a class does not make that function a member of the class.

Properties of Friend Function

- Not in the scope of the class, means it is not a member of the class.
- Since it is not in the scope of the class, it cannot be called from the object of that class.
- Can be declared anywhere inside the class, be it under the public or private access modifier, it will not make any difference.
- It cannot access the members directly by their names, it needs (object_name.member_name) to access any member.

The syntax for declaring a friend function inside a class is

```
class class_name
```

```
{
   friend return_type function_name(arguments);
};


return_type class_name::function_name(arguments)
{
   //body of the function
}
```

Friend Classes in C++

Friend classes are those classes that have permission to access private members of the class in which they are declared. The main thing to note here is that if the class is made friends of another class then it can access all the private members of that class.

The syntax for declaring a friend class inside a class is

```
class class_name
{
   friend class friend_class_name;
};
```

C++ Constructors

A constructor is a special member function with the same name as the class. The constructor doesn't have a return type. Constructors are used to initialize the objects of their class. Constructors are automatically invoked whenever an object is created.

Characteristics of Constructors in C++

- A constructor should be declared in the public section of the class.
- They are automatically invoked whenever the object is created.
- They cannot return values and do not have return types.
- It can have default arguments.

An example of how a constructor is used is:

```
#include <iostream>

using namespace std;
```

```
class Employee
{
public:
   static int count; // returns number of employees
   string eName;

   // Constructor
   Employee()
   {
      count++; // increases employee count every time an object is defined
   }

   void setName(string name)
   {
      eName = name;
   }

   static int getCount()
   {
      return count;
   }
};

int Employee::count = 0; // defining the value of count

int main()
{
   Employee Harry1;

   Employee Harry2;

   Employee Harry3;

   cout << Employee::getCount() << endl;

}
```

Output:

```
3
```

Parameterized and Default Constructors in C++

Parameterized constructors are those constructors that take one or more parameters. Default constructors are those constructors that take no parameters. This could have helped in the above example by passing the employee name at the time of definition only. That should have removed the setName function.

Constructor Overloading in C++

Constructor overloading is a concept similar to function overloading. Here, one class can have multiple constructors with different parameters. At the time of definition of an instance, the constructor, which will match the number and type of arguments, will get executed.

For example, if a program consists of 3 constructors with 0, 1, and 2 arguments and we pass just one argument to the constructor, the constructor which is taking one argument will automatically get executed.

Constructors with Default Arguments in C++

Default arguments of the constructor are those which are provided in the constructor declaration. If the values are not provided when calling the constructor, the constructor uses the default arguments automatically.

An example that shows declaring default arguments is:

```
class Employee{public:    Employee(int a, int b = 9);};
```

Copy Constructor in C++

A copy constructor is a type of constructor that creates a copy of another object. If we want one object to resemble another object, we can use a copy constructor. If no copy constructor is written in the program, the compiler will supply its own copy constructor.

The syntax for declaring a copy constructor is:

```
class class_name
```

```
{
   int a;


public:
   // copy constructor
   class_name(class_name &obj)
   {
      a = obj.a;
   }
};
```

C++ Encapsulation

Encapsulation is the first pillar of Object Oriented Programming. It means wrapping up data attributes and methods together. The goal is to keep sensitive data hidden from users.

Encapsulation is considered a good practice where one should always make attributes private for them to become non-modifiable until needed. The data is ultimately more secure as a result of this. Once members are made private, methods to access them or change them should be declared.

An example of how encapsulation is achieved is:

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


class class_name
{
private:
   int a;


public:
   void setA(int num)
```

```
  {
    a = num;
  }

  int getA()
  {
    return a;
  }
};

int main()
{
  class_name obj;
  obj.setA(5);
  cout << obj.getA() << endl;
}
```

Output:

```
5
```

C++ Inheritance

What is Inheritance in C++?

The concept of reusability in C++ is supported using inheritance. We can reuse the properties of an existing class by inheriting it and deriving its properties. The existing class is called the base class and the new class which is inherited from the base class is called the derived class.

The syntax for inheriting a class is:

```
// Derived Class syntax

class derived_class_name : access_modifier base_class_name
```

```
{
   // body of the derived class
}
```

Types of inheritance in C++

Single Inheritance

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class.

For example, we have two classes ClassA and ClassB. If ClassB is inherited from ClassA, it means that ClassB can now implement the functionalities of ClassA. This is single inheritance.

```
class ClassA
{
   // body of ClassA
};


// derived from ClassA
class ClassB : public ClassA
{
   // body of ClassB
};
```

Multiple Inheritance

Multiple inheritance is a type of inheritance in which one derived class is inherited from more than one base class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassC is inherited from both ClassA & ClassB, it means that ClassC can now implement the functionalities of both ClassA & ClassB. This is multiple inheritance.

```
class ClassA
{
   // body of ClassA
};
```

```
class ClassB
{
   // body of ClassB
};

// derived from ClassA and ClassB
class ClassC : public ClassA, public ClassB
{
   // body of ClassC
};
```

Hierarchical Inheritance

Hierarchical inheritance is a type of inheritance in which several derived classes are inherited from a single base class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassB and ClassC are inherited from ClassA, it means that ClassB and ClassC can now implement the functionalities of ClassA. This is hierarchical inheritance.

```
class ClassA
{
   // body of ClassA
};

// derived from ClassA
class ClassB : public ClassA
{
   // body of ClassB
};

// derived from ClassA
```

```
class ClassC : public ClassA
{
   // body of ClassC
};
```

Multilevel Inheritance

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassB is inherited from ClassA and ClassC is inherited from ClassB, it means that ClassB can now implement the functionalities of ClassA and ClassC can now implement the functionalities of ClassB. This is multilevel inheritance.

```
class ClassA
{
   // body of ClassA
};

// derived from ClassA
class ClassB : public ClassA
{
   // body of ClassB
};

// derived from ClassB
class ClassC : public ClassB
{
   // body of ClassC
};
```

Hybrid Inheritance

Hybrid inheritance is a combination of different types of inheritances.

For example, we have four classes ClassA, ClassB, ClassC, and ClassD. If ClassC is inherited from both ClassA and ClassB and ClassD is inherited from ClassC, it means that ClassC can now implement the functionalities of both ClassA and ClassB and ClassD can now implement the functionalities of ClassC. This is hybrid inheritance where both multilevel and multiple inheritances are present.

```
class ClassA

{

  // body of ClassA

};


class ClassB

{

  // body of ClassB

};


// derived from ClassA and ClassB

class ClassC : public ClassA, public ClassB

{

  // body of ClassC

};


// derived from ClassC

class ClassD : public ClassC

{

  // body of ClassD

};
```

C++ Access Modifiers

Public Access Modifier in C++

All the variables and functions declared under the public access modifier will be available for everyone. They can be accessed both inside and outside the class. Dot (.) operator is used in the program to access public data members directly.

Private Access Modifier in C++

All the variables and functions declared under a private access modifier can only be used inside the class. They are not permissible to be used by any object or function outside the class.

Protected Access Modifiers in C++

Protected access modifiers are similar to the private access modifiers but protected access modifiers can be accessed in the derived class whereas private access modifiers cannot be accessed in the derived class.

A table is shown below which shows the behavior of access modifiers when they are derived from public, private, and protected.

	Public Derivation	**Private Derivation**	**Protected Derivation**
Private members	Not inherited	Not inherited	Not inherited
Protected Members	Protected	Private	Protected
Public Members	Public	Private	Protected

- If the class is inherited in public mode then its private members cannot be inherited in child class.
- If the class is inherited in public mode then its protected members are protected and can be accessed in child class.
- If the class is inherited in public mode then its public members are public and can be accessed inside the child class and outside the class.
- If the class is inherited in private mode then its private members cannot be inherited in child class.

- If the class is inherited in private mode then its protected members are private and cannot be accessed in child class.
- If the class is inherited in private mode then its public members are private and cannot be accessed in child class.
- If the class is inherited in protected mode then its private members cannot be inherited in child class.
- If the class is inherited in protected mode then its protected members are protected and can be accessed in the child class.
- If the class is inherited in protected mode then its public members are protected and can be accessed in the child class.

C++ Polymorphism

Polymorphism in C++

Poly means several and morphism means form. Polymorphism is something that has several forms or we can say it as one name and multiple forms. There are two types of polymorphism:

- Compile-time polymorphism
- Run time polymorphism

Compile Time Polymorphism

In compile-time polymorphism, it is already known which function will run. Compile-time polymorphism is also called early binding, which means that you are already bound to the function call and you know that this function is going to run.

There are two types of compile-time polymorphism:

- **Function Overloading**

This is a feature that lets us create more than one function and the functions have the same names but their parameters need to be different. When function overloading is implemented in a program and function calls are made, the compiler knows which functions to execute.

- **Operator Overloading**

This is a feature that lets us define operators working for some specific tasks. Using the operator +, we can add two strings by concatenating and add two numerical values arithmetically at the same time. This is operator overloading.

Run Time Polymorphism

With run-time polymorphism, the compiler has no idea what will happen when the code is executed. Run time polymorphism is also called late binding. The run-time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from virtual functions.

Virtual Functions in C++

A member function in the base class that is declared using a virtual keyword is called a virtual function. They can be redefined in the derived class. A function that is in the parent class but redefined in the child class is called a virtual function.

For a function to become virtual, it should not be static.