Object-Oriented Programming

OOP stands for Object-Oriented Programming.

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:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

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

C++ Classes/Objects:

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating.

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.

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.

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute
    string myString;  // Attribute
};
```

Object:

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc. In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality. Object is a runtime entity, it is created at runtime.

```
MyClass myObj;  // Create an object of MyClass
  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";
```

Class Methods:

Methods are functions that belongs to the class.

There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

In the following example, we define a function inside the class, and we name it "myMethod".

Inside Example:

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod() {  // Method/function defined inside the class
      cout << "Hello World!";
    }
};


int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Outside Example:

```
class MyClass {        // The class
  public:              // Access specifier
    void myMethod();   // Method/function declaration
};
// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}
int main() {
```

```
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

Eg.

```
class Student {
public:
//Declaration of state/Properties.
string name;  //Instance variable.
int rollNo;  // Instance variable.
static int age; // Static variable.
// Declaration of Actions.
void display() { // Instance method.
// method body.
}
};
```

REAL LIFE EXAMPLE:

Object: Person

State/Properties:

Black Hair Color: hairColor = "Black";

5.5 feet tall: height = 5.5;

Weighs 60 kgs: weight = 60;

Behaviour/Action:

Eat: eat()

Sleep: sleep(5)

Run: run(0.5)

Encapsulation

Encapsulation is defined as binding together the data and the functions that manipulate them.

Example:

Consider, in a company, there are different sections like the accounts section, finance section, sales section, etc. 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.

```
// Program to calculate the area of a rectangle
#include <iostream>
using namespace std;
class Rectangle {
 public
 int length;
 int breadth;
 Rectangle(int len, int brth) : length(len), breadth(brth) {}
 int getArea() {
 return length * breadth;
 }
};
int main() {
 Rectangle rect(8, 6);
```

```
 cout << "Area = " << rect.getArea();
 return 0;
}
```

Abstraction:

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Example:

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 an accelerator, brakes, etc. in the car.

```
#include <iostream>
using namespace std;
class implementAbstraction {
private:
 int a, b;
public:
 void set(int x, int y)
 {
 a = x;
 b = y;
 }
 void display()
 {
 cout << "a = " << a << endl;
 cout << "b = " << b << endl;
```

```
 }
};
int main()
{
 implementAbstraction obj;
 obj.set(10, 20);
 obj.display();
 return 0;
}
```

Polymorphism:

The word polymorphism means having many forms.we can define polymorphism as the ability of a message to be displayed in more than one form.

Example:

A person 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 possesses different behavior in different situations.

C++ supports operator overloading and function overloading.

 Operator Overloading: The process of making an operator exhibit different behaviors in different instances is known as operator overloading.

 Function Overloading: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

Function overloading

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Value {
public:
 void func(int x)
 {
 cout << "value of x is " << x << endl;
 }
 void func(double x)
 {
 cout << "value of x is " << x << endl;
 }
 void func(int x, int y)
 {
 cout << "value of x and y is " << x << ", " << y
 << endl;
 }
};
int main()
{
 Value obj1;
 obj1.func(7);
 obj1.func(9.132);
 obj1.func(85, 64);
 return 0;
}
```

Operator Overloading:

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

```
class Complex {
private:
 int real, imag;
public:
 Complex(int r = 0, int i = 0)
 {
 real = r;
 imag = i;
 }
 Complex operator+(Complex const& obj)
 {
 Complex res;
 res.real = real + obj.real;
 res.imag = imag + obj.imag;
 return res;
 }
 void print() { cout << real << " + i" << imag << endl; }
};
int main()
{
 Complex c1(10, 5), c2(2, 4);
 Complex c3 = c1 + c2;
 c3.print();
}
```

Function overriding

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Animal {
public:
 string color = "Black";
};
class Dog : public Animal {
public:
 string color = "Grey";
};
int main(void)
{
 Animal d = Dog();
 cout << d.color;
}
```

Inheritance:

The capability of a class to derive properties and characteristics from another class is called Inheritance.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.

```
class Vehicle {
 public:
 string brand = "Ford";
 void honk() {
 cout << "Tuut, tuut! \n" ;
 }
};
class Car: public Vehicle {
 public:
 string model = "Mustang";
```

```
};
int main() {
 Car myCar;
 myCar.honk();
 cout << myCar.brand + " " + myCar.model;
 return 0;
}
```

Types Of Inheritance:-

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance

Dynamic Binding:

In dynamic binding, the code to be executed in response to the function call is decided at runtime.

```
#include <iostream>
using namespace std;
class Trial {
public:
 void call_Function()
 {
 print();
 }
 void print()
 {
 cout << "Printing the Base class Content" << endl;
 }
```

```
};
class Trial2 : public Trial
{
public:
 void print()
 {
 cout << "Printing the Derived class Content"
 << endl;
 }
};
int main()
{
 Trial try;
 try.call_Function();
 Trial2 try2;
 try.call_Function();
 return 0;
}
```

Message Passing:

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

Access Specifiers:

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class

- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

```
class MyClass {
  public:               // Public access specifier
    int x;        // Public attribute
  private:      // Private access specifier
    int y;         // Private attribute
};
int main() {
  MyClass myObj;
  myObj.x = 25;               // Allowed (public)
  myObj.y = 50;               // Not allowed (private)
  return 0;
}
```

Default Copying

By default, objects can be copied. In particular, a class object can be initialized with a copy of an object of its class. For example:

Date d1 = my_birthday; // initialization by copy

Date d2 {my_birthday}; // initialization by copy

By default, the copy of a class object is a copy of each member.

If that default is not the behavior wanted for a class X, a more appropriate behavior can be provided using overloading.

Static Members of a C++ Class

We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Eg:

```
class Box {
public:
static int objectCount;
};
// Initialize static member of class Box
int Box::objectCount = 0;
```

Static Function Members

```
class Box {
public:
static int objectCount;
static int getCount() {
return length;
}
private:
double length; // Length of a box
};
Box::getCount();
```

Passing and Returning Objects in C++

In C++ we can pass class objects as arguments and also return them from a function the same way we pass and return other variables.

Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

```
fun(object_name);
```

Passing an Object as argument

```
class Example {
public:
int a;
void add(Example E)
{
a = a + E.a;
}
};
```

Returning Object as argument

```
class Example {
public:
int a;
Example add(Example Ea, Example Eb) {
Example Ec;
Ec.a Ec.a+ Ea.a + Eb.a;
return Ec;
}
};
```

Friend Class and Function in C++:

A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node. A friend class can access private and protected members of other class which it is declared as friend.

We can declare a friend class in C++ by using the friend keyword.

Syntax:

```
friend class class_name;    // declared in the base class
```

Example:

```
class A {
private:
        int a;
public:
       A(){
       a = 0;
      }
friend class B: // Friend Class
};
class B {
private:
       int b;
public:
      void showA(A&x) {
     cout << x.a;
     }
};
```

Pointers

Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax:

```
datatype *var_name;
int *ptr;
```