OOPs Definition

Definition

Object-Oriented Programming is basically a programming style that we used to follow in modern programming. It primarily revolves around classes and objects. Object-Oriented programming or OOPs refers to the language that uses the concept of class and object in programming. The popular object-oriented programming languages are c++, java, python, PHP, c#, etc. The main objective of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc.

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.

Class

A class is a logical entity used to define a new data type. A class is a user-defined type that describes what a particular kind of object will look like. Thus, a class is a template or blueprint for an object. A class contains variables, methods, and constructors.

Classes are very useful in programming. Consider the example of where you don't want to use just one car but 100 cars. Rather than describing each one in detail from scratch, you can use the same car class to create 100 objects of the type 'car'. You still have to give each one a name and other properties, but the basic structure of what a car looks like is the same.

The car class would allow the programmer to store similar information unique to each car (different models, maybe different colors, etc.) and associate the appropriate information with each car.

Syntax to define a class:-

```
class class_name{
// class body
        //properties
        //methods
};
```

Here,

- class: class keyword is used to create a class in C++.
- class_Name: The name of the class.
- class body: Curly braces surround the class body.
- After closing curly braces, a semicolon(;) is used.

Object

An object is an instance of a Class. It is an identifiable entity with some characteristics and behavior. Objects are the basic units of object-oriented programming. It may be any real-world object like a person, chair, table, pen, animal, car, etc.
A simple example of an object would be a car. Logically, you would expect a car to have a model number or name. This would be considered the property of the car. You could also expect a car to be able to do something, such as starting or moving. This would be considered a method of the car.

Code in object-oriented programming is organized around objects. Once you have your objects, they can interact with each other to make something happen.

You need to have a class before you can create an object. When a class is defined, no memory is allocated, but memory is allocated when it is instantiated (i.e., an object is created).

Syntax to create an object in C++:

```
class_name objectName;
```

Syntax to create an object dynamically in C++:

```
class_name * objectName = new class_name();
```

Here,

- objectName: It is the name of the object created by class_name.

The class's default constructor is called, and it dynamically allocates memory for one object of the class. The address of the memory allocated is assigned to the pointer, i.e., objectName.

Features of OOPs:-

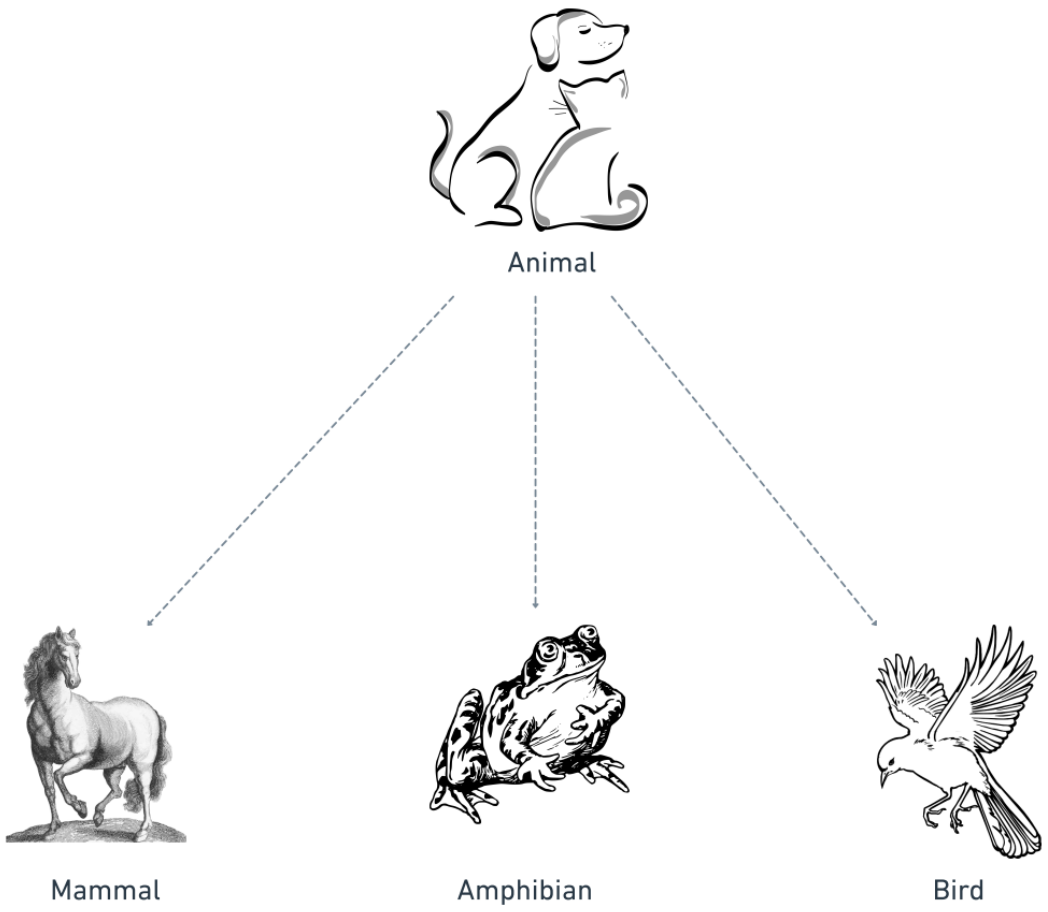
Four major object-oriented programming features make them different from non-OOP languages:

- **Abstraction** is the property by virtue of which only the essential details are displayed to the user.
- **Inheritance** allows you to create class hierarchies, where a base class gives its behavior and attributes to a derived class.
- **Polymorphism** ensures that it will execute the proper method based on the calling object's type.
- **Encapsulation** allows you to control access to your object's state while making it easier to maintain or change your implementation at a later date.

Real-world class modeling

Realworld class modeling example:-

Let’s take a real-world example of Animal as a class to understand concepts better. And Mammals, birds, and amphibians as objects of the class.



Creating a class Animal and objects mammal, amphibian and bird:-

```
#include <iostream>
Using namespace std;
// creating Animal class
class Animal{
        bool gives_birth;
        bool lay_egg;
        bool live_in_ground;
        bool live_in_water;
        bool have_wings;
};

int main(){
        // creating an object of animal class
        Animal mammal;

        //assign values to instance variables
        mammal.gives_birth = true;
        mammal.lay_egg = false;
        mammal.live_in_ground = true;
        mammal.live_in_water = false;
        mammal.have_wings = false;

        Animal amphibian;
        amphibian.gives_birth = false;
        amphibian.lay_egg = true;
        amphibian.live_in_ground = true;
        amphibian.live_in_water = true;
        amphibian.have_wings = false;

        Animal bird;
        bird.gives_birth = false;
        bird.lay_egg = true;
        bird.live_in_ground = true;
        bird.live_in_water = false;
        bird.have_wings = true;
}
```

We all know animals are a creature of God, every animal is different from each other, but they also possess some unique properties. Here we create a class Animal and define some animal characters (properties) that may be shared for different kinds of animals. We defined all the properties for each object, like whether they give birth or not, whether they live in water, etc.

Here, Animal class provides a template or blueprint for creating objects (mammal, bird, and amphibian).

Why do we need object-oriented programming?

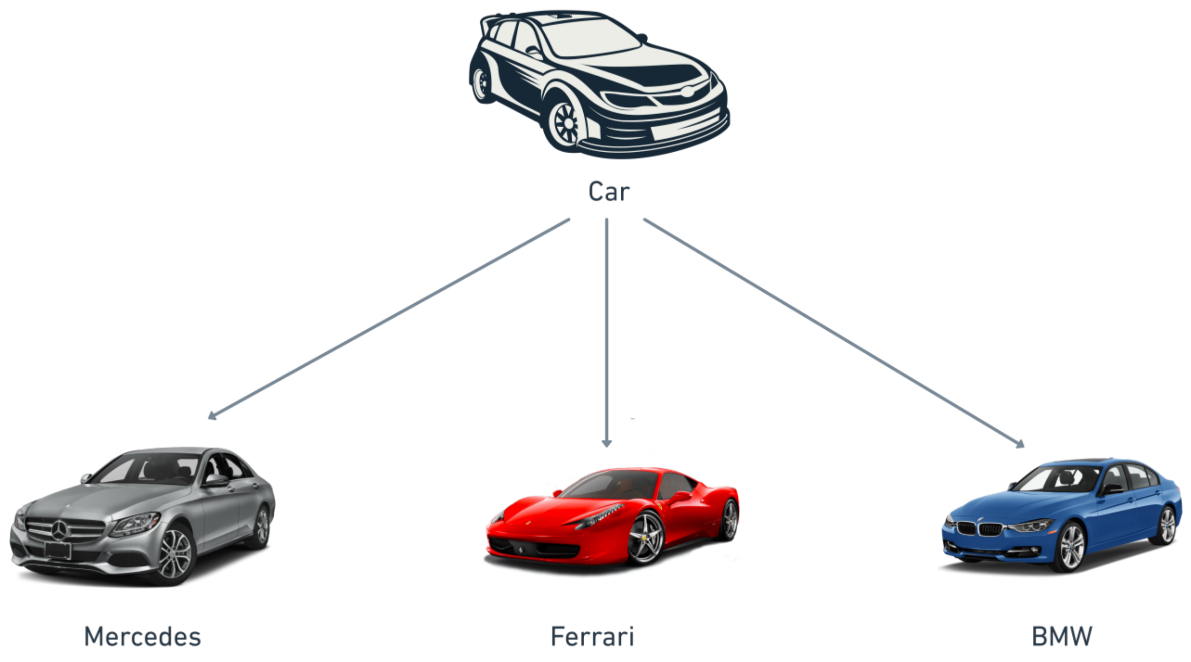
- To make the development and maintenance of projects more effortless.
- To provide the feature of data hiding that is good for security concerns.
- We can solve real-world problems if we are using object-oriented programming.
- It ensures code reusability.
- It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.

Example of OOPs in the Industry

Example of OOPs using Cars:-

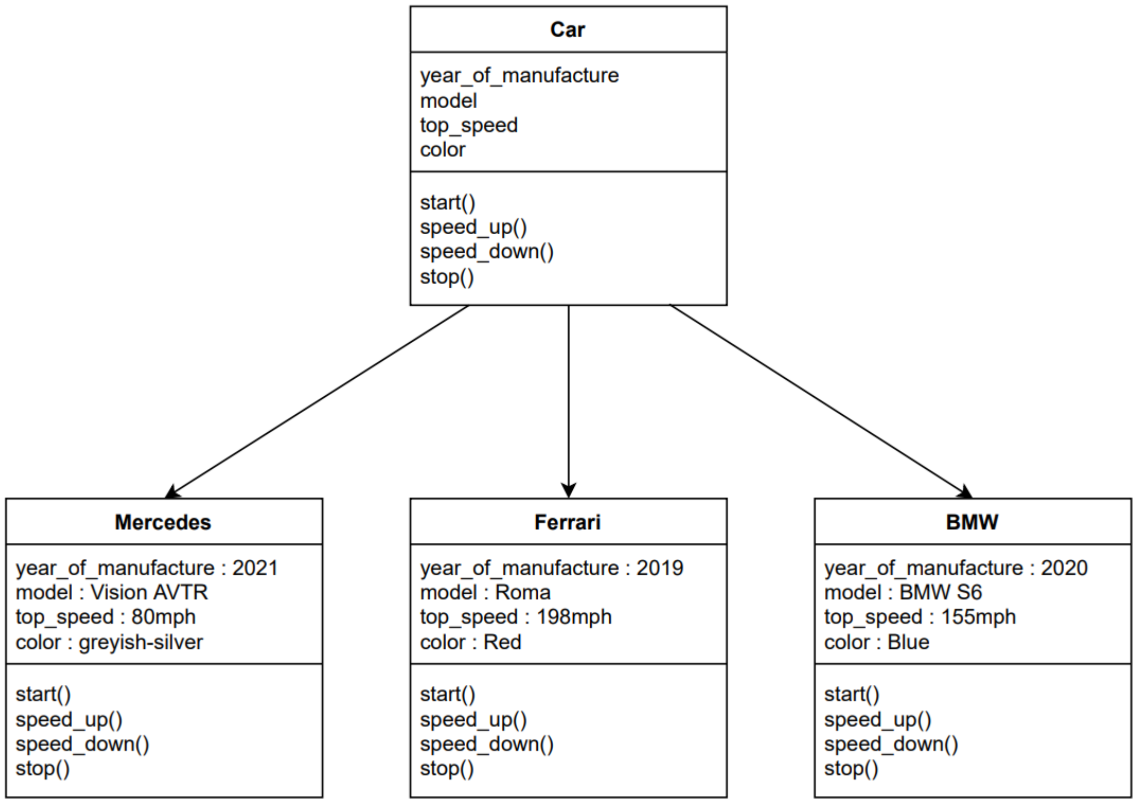
Consider the example of where you don't want to use just one car but 100 cars. Rather than describing each one in detail from scratch, you can use the same car class to create 100 objects of the type 'car'. You still have to give each one a name and other properties, but the basic structure of what a car looks like is the same.

Here we will make Car class, and it will work as a basic template for other objects. We will make car class objects (Ferrari, BMW, and Mercedes). Each Car Object will have its own, Year of Manufacture, model, Top Speed, color, Engine Power, efficiency, etc.



The car class would allow the programmer to store similar information unique to each car (different models, colors, top speeds, etc.) and associate the appropriate information.

Understanding example using flowchart:-



Class

Class

A class is a logical entity used to define a new data type. A class is a user-defined type that describes what a particular kind of object will look like. A class contains variables(data members), methods, and constructors.

Class is a blueprint or a set of instructions to build a specific type of object. It is a fundamental concept of Object-Oriented Programming which revolves around real-life entities. Class determines how an object will behave and what the object will contain.

Data encapsulation is supported with “class”. The class consists of both data and functions. The data in a class is called a member, while functions in the class are called methods.

Data Members:- The variables which are declared in any class by using any fundamental data types (like int, char, float, etc.) or derived data types (like class, structure, pointer, etc.) are known as Data Members.

Methods:- A method is the equivalent of a function in object-oriented programming that is used inside classes. The methods are the actions that perform operations. A method accepts parameters as arguments, manipulates these, and then produces an output when the method is called on an object.

Constructor:- Constructors are special class functions that perform the initialization of every object. In C++, the constructor is automatically called when an object is created. It is a special method of the class because it does not have any return type. It has the same name as the class itself.

Syntax to define a class:-

```
class class_name{
        //class body
```

```
                //data_members
                //constructor (optional)
                //methods
};
```

Here,

- ❖ class: class keyword is used to create a class in C++.
- ❖ class_name: The name of the class.
- ❖ class body: Curly braces surround the class body.
- ❖ After closing curly braces, a semicolon(;) is used.

Classes are very useful in programming. Consider the example of where you don't want to use just one Smartphone but 100 smartphones. Rather than describing each one in detail from scratch, you can use the same smartphone class to create 100 objects of the type 'smartphones'. You still have to give each one a name and other properties, but the basic structure of what a smartphone looks like is the same.

The smartphone class would allow the programmer to store similar information unique to each car (different models, maybe different colors, etc.) and associate the appropriate information with each smartphone.

Example of smartphone class:-

```
class smartphone{
  //class body

    //Data Members(Properties)
    string model;
    int year_of_manufacture;
    bool _5g_supported;

    //Constructor
    smartphone(string mod, int manu, bool _5g_supp){
      //initialzing data members
      model = mod;
      year_of_manufacture = manu;
      _5g_supported = _5g_supp;
    }

    //methods
```

```
        void print_details(){
            cout << "Model : " << model << endl;
            cout << "Year of Manufacture : " << year_of_manufacture << endl;
            cout << "5g Supported : " << _5g_supported << endl;
        }
};
```

Object

Object

An object is an instance of a Class. It is an identifiable entity with some characteristics and behavior. To access the members defined inside the class, we need to create the object of that class. Objects are the basic units of object-oriented programming. It may be any real-world object like a person, chair, table, pen, animal, car, etc.

Code in object-oriented programming is organized around objects. Once you have your objects, they can interact with each other to make something happen.

Syntax to create an object in C++:

```
class_name objectName;
```

Syntax to create an object dynamically in C++:

```
class_name * objectName = new class_name();
```

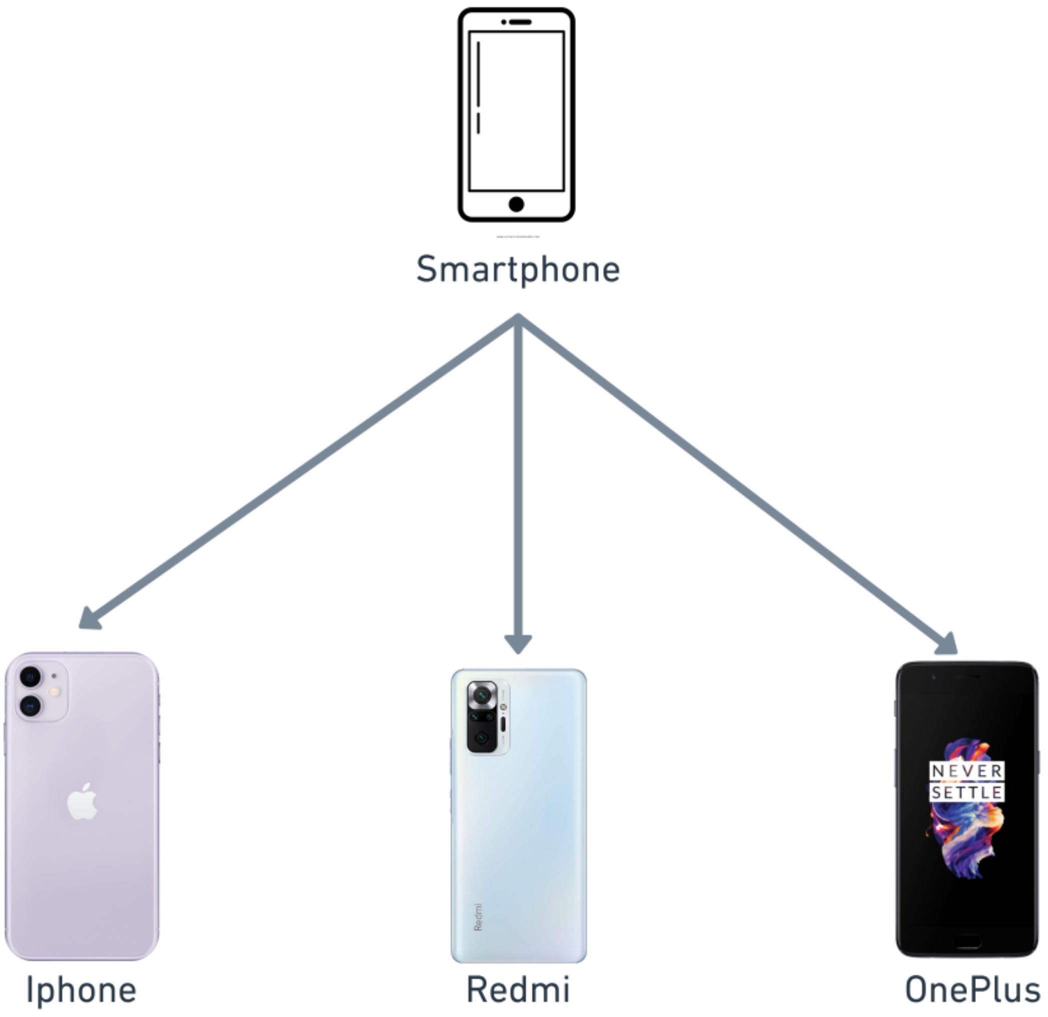
Here,

❖ objectName: It is the name of the object created by class_name.

The class's default constructor is called, and it dynamically allocates memory for one object of the class. The address of the memory allocated is assigned to the pointer, i.e., objectName.

Explaining object using Smartphone example with code:-

A simple example of an object would be a smartphone. Logically, you would expect a smartphone to have a model number or name. This would be considered the property of the smartphone. You could also expect a smartphone to do something, such as to send SMS, etc. This would be considered a method of the smartphone.



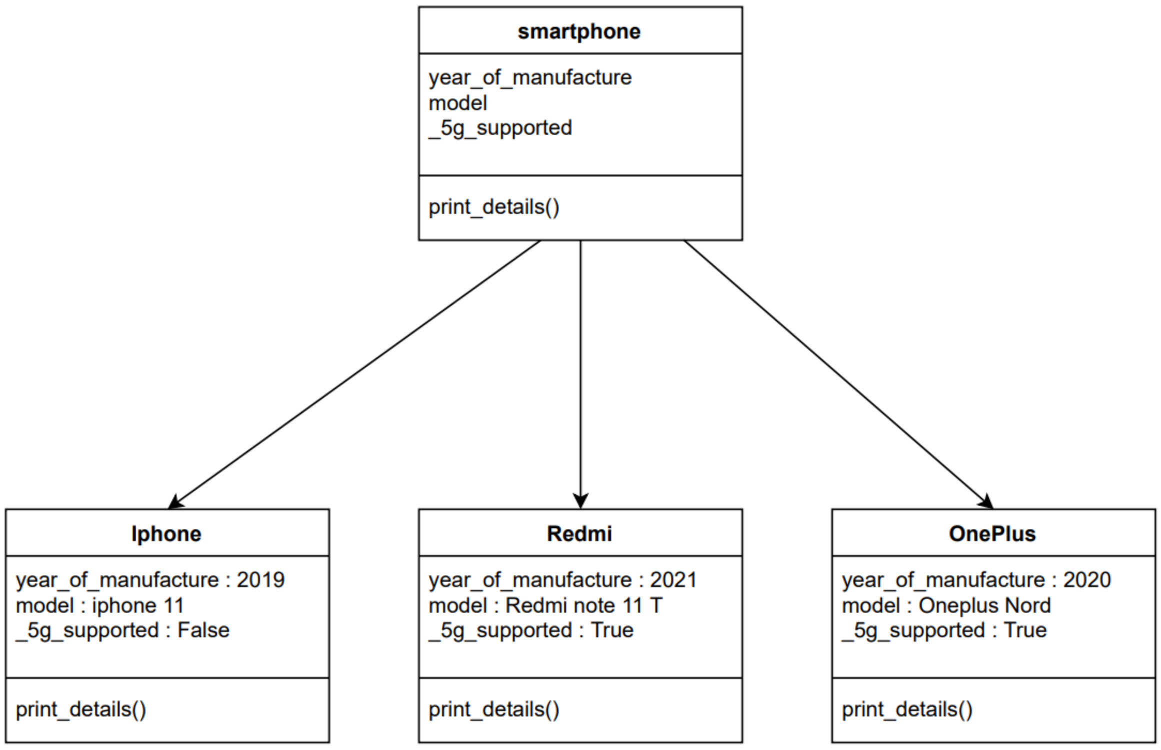
We have created a smartphone class earlier in the class module, and Now we will use that same class to make objects.

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

//creating class
class smartphone{
    //class body

        //Data Members(Properties)
        string model;
        int year_of_manufacture;
        bool _5g_supported;

        //Constructor
        smartphone(string model_string, int manufacture, bool _5g_){
            //initialzing data members
            model = model_string;
            year_of_manufacture = manufacture;
            _5g_supported = _5g_;
        }

        //methods
        void print_details(){
            cout << "Model : " << model << endl;
            cout << "Year of Manufacture : " << year_of_manufacture << endl;
            cout << "5g Supported : " << _5g_supported << endl;
        }
};


int main(){
        //creating objects of smartphone class
        smartphone iphone("iphone 11", 2019, false );
        smartphone redmi("redmi note 11 t", 2021, true );
        smartphone oneplus("oneplus nord", 2020, true );

        //accessing class variables
        int iphone_manufacture_date = iphone.year_of_manufacture;
        bool redmi_support_5g = redmi._5g_supported;
        string oneplus_model = oneplus.model;

        //calling methods on objects
        iphone.print_details();
        redmi.print_details();
        oneplus.print_details();
}
```

★ To create an object of a smartphone, specify the class name, followed by the object name.
★ To access the class attributes or data members (like model), use the dot syntax (.) on the object followed by the attribute name.
★ To call any method (print_details()) of class, use the dot syntax (.) on the object followed by the method name.

You need to have a class before you can create an object. When a class is defined, no memory is allocated, but memory is allocated when it is instantiated (i.e., an object is created).

Graphical Representation of smartphone class and its object:-



Access Specifiers

Access Specifier:-

Access Specifiers in a class are used to assign access to the class members. It sets some restrictions on the class members from accessing the outside functions directly. Access specifiers have a vital role in securing data from unauthorized access.

It allows us to determine which class members are accessible to other classes and functions and which are not.
There are three types of access modifiers available in C++:

- Public
- Private
- Protected

➔ **Public:** All the class members with a public modifier can be accessed from anywhere(inside and outside the class).

```
class person{
    public:
    string name;
};
```

➔ **Private:** All the class members with a private modifier can only be accessed by the member function inside the class.

```
class person{
    private:
    int fb_password;
};
```

➔ **Protected:** The access level of a protected modifier is within the class and outside the class through child class (or subclass). If you do not make the child class, it cannot be accessed outside the class.

```
class person{
    protected:
    string assets;
};
```

- ➔ **By default**, in C++, all class members are private if you don't specify an access specifier.

```
class person{
    int name; //by default, it is a private data member
};
```

Example using smartphone class:-

```
class smartphone{
    //Data members
    string model;  // by default private

    public:
    int year_of_manufacture; // public data member

    protected:
    string company_name;  // protected data member

    private:
    int password  // private data member

    //methods
    private:
    void unlock_lockscreen(){
        //private method
    }

    public:
    void call(){
        //public method
    }

    protected:
    void about_phone(){
        //protected method
    }
};
```

Scope Table:-

	Inside class	Child (or sub-class)	Outside class
Public	✔	✔	✔
Protected	✔	✔	❌
Private	✔	❌	❌

Interview Questions

1. **Why do we use OOPs?**
    - It gives clarity in programming and allows simplicity in solving complex problems.
    - Data and code are bound together by encapsulation.
    - Code can be reused, and it reduces redundancy.
    - It also helps to hide unnecessary details with the help of Data Abstraction.
    - Problems can be divided into subparts.
    - It increases the readability, understandability, and maintainability of the code.
2. **What are the differences between the constructor and the method?**

Constructor	Method
It is a block of code that initializes a newly created object.	It is a group of statements that can be called at any point in the program using its name to perform a specific task.
It has the same name as the class name.	It should have a different name than the class name.
It has no return type.	It needs a valid return type if it returns a value; otherwise void.
It is called implicitly at the time of object creation	It is called explicitly by the programmer by making a method call
If a constructor is not present, a default constructor is provided by Java	In the case of a method, no default method is provided.

3. **What are the main features of OOPs?**
    - Inheritance
    - Encapsulation
    - Polymorphism
    - Data Abstraction
4. **The disadvantage of OOPs?**
    - Requires pre-work and proper planning.
    - In certain scenarios, programs can consume a large amount of memory.
    - Not suitable for a small problem.
    - Proper documentation is required for later use.
5. **What is the difference between class and structure?**

   Class: User-defined blueprint from which objects are created. It consists of methods or sets of instructions that are to be performed on the objects.
   Structure: A structure is basically a user-defined collection of variables of different data types.

6. **What is the difference between a class and an object?**

Class	Object
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a car.	Objects of the class car can be BMW, Mercedes, Ferrari, etc.

Constructor

Constructor-

A constructor is a special member function automatically called when an object is created. In C++, the constructor is automatically called when an object is created. It is a special class method because it does not have any return type. It has the same name as the class itself.

A constructor initializes the class data members with garbage value if we don't put any value to it explicitly.

The constructor must be placed in the public section of the class because we want the class to be instantiated anywhere. For every object in its lifetime constructor is called only once at the time of creation.

Example:

```
class class_name{
    int data_member1;
    string data_member2;

    //creating constructor
    public:
    class_name(){
        // initialize data members with garbage value
    }
};
```

Here, the function class_name() is a constructor of the class 'class_name'. Notice that the constructor

- ❖ has the same name as the class,
- ❖ does not have any return type, and
- ❖ it is public

If we do not specify a constructor, the C++ compiler generates a default constructor for an object (which expects no parameters and has an empty body).

Types of Constructors:

There are three types of constructors in C++:

- ★ Default constructor
- ★ Parameterized Constructor
- ★ Copy Constructor

Default constructor:-

A constructor that doesn't take any argument or has no parameters is known as a default constructor. In the example above, class_name() is a default constructor.

Syntax:

```
class class_name{
    int data_member1;
    string data_member2;

    //default constructor
    public:
    class_name(){
        // initializing data members with their default values
        data_member1 = 69;
        data_member2 = "Coding Ninjas";
    }
};
```

Here, the class_name() constructor will be called when the object is created. This sets the data_member1 variable of the object to 69 and the data_member2 variable of the object to “Coding Ninjas”.

Note: If we have not defined a constructor in our class, the C++ compiler will automatically create a default constructor with an empty code and no parameters, which will initialize data members with garbage values.

When we write our constructor explicitly, the inbuilt constructor will not be available for us.

Parameterized Constructor:-

This is another type of Constructor with parameters. The parameterized constructor takes its arguments provided by the programmer. 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 defining the constructor's body, use the parameters to initialize the object.

Using this Constructor, you can provide different values to data members of different objects by passing the appropriate values as arguments.

Syntax:

```
class class_name{
    int data_member1;
    string data_member2;

    //parameterized constructor
    public:
    class_name(int num, string str){
        // initializing data members with values provided
        data_member1 = num;
        data_member2 = str;
    }
};
```

Here, we have created a parameterized constructor class_name() that has 2 parameters: int num and string str. The values contained in these parameters are used to initialize the member variables data_member1 and data_member2.

Copy Constructor:-

These are a particular type of constructor that takes an object as an argument and copies values of one object's data members into another object. We pass the class object into another object of the same class in this constructor. As the name suggests, you Copy means to copy the values of one Object into another Object of Class. This is used for Copying the values of a class object into another object of a class, so we call them Copy constructor and for copying the values.

We have to pass the object's name whose values we want to copy, and when we are using or passing an object to a constructor, we must use the & ampersand or address operator.

Syntax:

```
class class_name{
    int data_member1;
    string data_member2;

    //copy constructor
    public:
    class_name(class_name &obj){
        // copies data of the obj parameter
        data_member1 = obj.data_member1;
        data_member2 = obj.data_member2;
    }
};
```

In this program, we have used a copy constructor to copy the contents of one object of the class 'class_name' to another. The code of the copy constructor is:

```
class_name(class_name &obj){
        // copies data of the obj parameter
        data_member1 = obj.data_member1;
        data_member2 = obj.data_member2;
}
```

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a memberwise copy between objects.

Example using smartphone class:

```
class smartphone{

    //Data Members(Properties)
    string model;
    int year_of_manufacture;
    bool _5g_supported;

    public:
    //default constructor
    smartphone(){
        model = "unknown";
```

```cpp
        year_of_manufacture = 0;
        _5g_supported = false;
    }
    //parameterized constructor
    smartphone(string model_string, int manufacture, bool _5g_){
        //initialising data members
        model = model_string;
        year_of_manufacture = manufacture;
        _5g_supported = _5g_;
    }

    // copy constructor
    smartphone(smartphone &obj){
        // copies data of the obj parameter
        model = obj.model;
        year_of_manufacture = obj.year_of_manufacture;
        _5g_supported = obj._5g_supported;
    }
};


int main(){
    //creating objects of smartphone class

    // using default constructor
    smartphone unknown;

    // using parameterized constructor
    smartphone iphone("iphone 11", 2019, false );

    // using copy constructor
    smartphone iphone_2(iphone);
}
```

Constructor Overloading

Constructor Overloading:-

Constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

As there is a concept of function overloading, similarly constructor overloading is applied when we overload a constructor more than a purpose.

The declaration is the same as the class name, but there is no return type as they are constructors.

The criteria to overload a constructor is to differ the number of arguments or the type of arguments. The corresponding constructor is called depending on the number and type of arguments passed.

Example using smartphone class:

```
class smartphone{

    //Data Members(Properties)
    string model;
    int year_of_manufacture;
    bool _5g_supported;

    public:
    //constructor with 0 parameter
    smartphone(){
        model = "unknown";
        year_of_manufacture = 0;
        _5g_supported = false;
    }

    //constructor with 2 parameter
    smartphone(string model_string, bool _5g_){
        model = model_string;
        _5g_supported = _5g_;
    }

    //constructor with 3 parameter
    smartphone(string model_string, int manufacture, bool _5g_){
        //initialising data members
        model = model_string;
        year_of_manufacture = manufacture;
        _5g_supported = _5g_;
    }
};


int main(){
    //creating objects of smartphone class

    // using constructor with 0 parameter
    smartphone unknown;

    //using constructor with 0 parameter
    smartphone redmi("Note 7 Pro", false);

    // using constructor with 3 parameter
    smartphone iphone("iphone 11", 2019, false );
}
```

Destructor

Destructor:-

A destructor is a special member function that works just opposite to a constructor; unlike constructors that are used for initializing an object, destructors destroy (or delete) the object. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

```
~class_name()
{
   //Some code
}
```

Like the constructor, the destructor name should exactly match the class name. A destructor declaration should always begin with the tilde(~) symbol, as shown in the syntax above.

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.

Example:-

```
#include <iostream>
using namespace std;
class Guided_path{
    public:
    //Constructor
    Guided_path ()
    {
        cout << "Constructor is called" << endl;
        cout<<"Welcome to Guided Path"<< endl;
    }
    //Destructor
    ~Guided_path ()
    {
        cout<< "Happy Learning"<< endl;
        cout << "Destructor is called" << endl;
    }
};

int main ()
{
  //Object created
  Guided_path obj;
  // at the end object destructed
}

Output:
Constructor is called
Welcome to Guided Path
Happy Learning
Destructor is called
```

When is a destructor called?

A destructor function is called automatically when:

➔ the object goes out of scope
➔ the program ends
➔ a scope (the { } parenthesis) containing local variable ends.
➔ a delete operator is called

Destructor rules

❖ The name should begin with a tilde sign(~) and match the class name.
❖ There cannot be more than one destructor in a class.
❖ Unlike constructors that can have parameters, destructors do not allow any parameter.
❖ They do not have any return type, not even void. I
❖ A destructor should be declared in the public section of the class.
❖ The programmer cannot access the address of the destructor.
❖ It has no return type, not even void.
❖ When you do not specify any destructor in a class, the compiler generates a default destructor and inserts it into your code.

Interview Questions

Interview Questions:-

1. **Does C++ compiler create a default constructor when we write our own?**

In C++, compiler by default creates a default constructor for every class. But, if we define our own constructor, compiler doesn't create the default constructor.

2. **Explain constructor in C++**

A constructor is a special member function automatically called when an object is created. A constructor initializes the class data members with garbage value if we don't put any value to it explicitly.

3. **What do you mean by constructor overloading?**

The concept of having more than one constructor with different parameters to perform a different task is known as constructor overloading.

4. **Explain Destructor in C++**

A destructor is a special member function that works just opposite to a constructor; unlike constructors that are used for initializing an object, destructors destroy (or delete) the object. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

5. **What is a copy constructor?**

These are a particular type of constructor that takes an object as an argument and copies values of one object's data members into another object. In this constructor, we pass the class object into another object of the same class.

6. **How many types of constructors are there?**

There are three types of constructors in C++:

- Default constructor
- Parameterized Constructor
- Copy Constructor

7. **When should the destructor use delete to free the memory?**

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.

8. **What is the return type of constructor and destructor?**

They have no return type, not even void.

this Pointer

this Pointer-

this pointer holds the address of the current object. In simple words, you can say that this pointer points to the current object of the class.

There can be three main usages of this keyword in C++.

- It can be used to refer to a current class instance variable.
- It can be used to pass the current object as a parameter to another method.
- It can be used to declare indexers.

Let’s take an example to understand this concept.

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

class mobile{
    string model;
    int year_of_manufacture;

    public:
    void set_details(string model, int year_of_manufacture){
        this->model = model;
        this->year_of_manufacture = year_of_manufacture;
    }

    void print(){
        cout << this->model << endl;
        cout << this->year_of_manufacture << endl;
    }
};

int main()
{
    mobile redmi;
    redmi.set_details("Note 7 Pro", 2019);
    redmi.print();
}
```

```
Output:
```

```
Note 7 Pro
2019
```

Here you can see that we have two data members model and year_of_manufacture. In member function set_details(), we have two local variables with the same name as the data members' names. Suppose you want to assign the local variable value to the data members. In that case, you won't be able to do until unless you use this pointer because the compiler won't know that you are referring to the object's data members unless you use this pointer. This is one of example where you must use this pointer.

Shallow and Deep Copy

Shallow Copy-

An object is created by simply copying the data of all variables of the original object. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred.

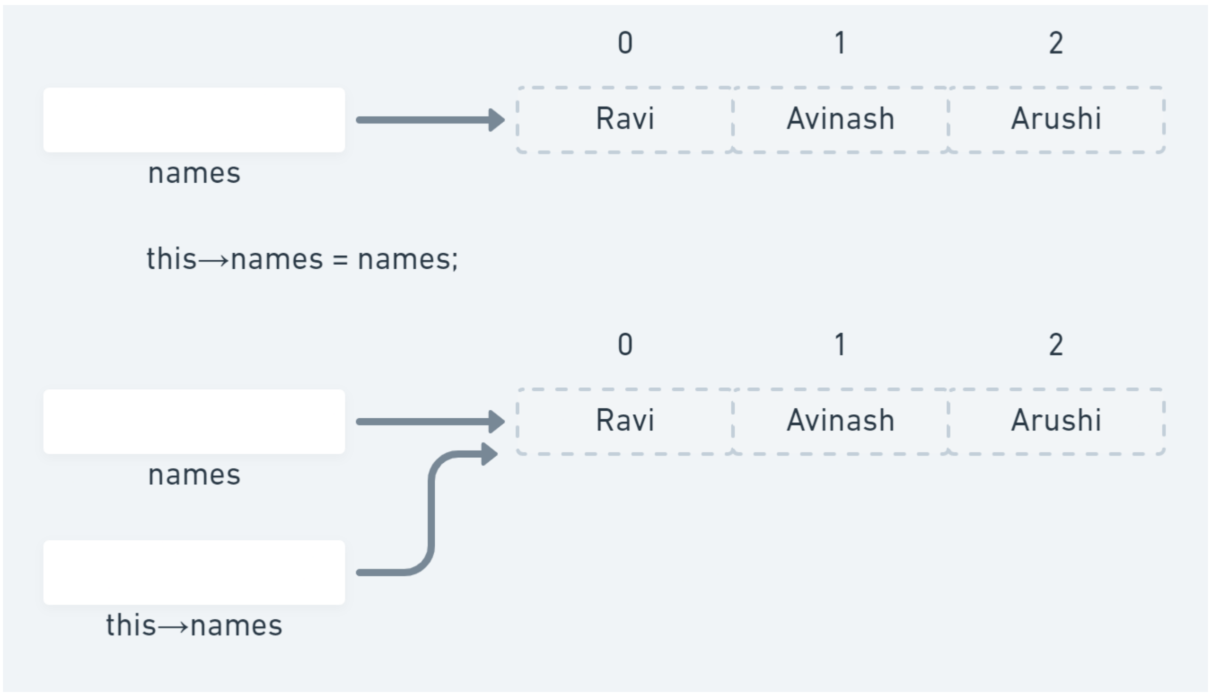
Since both objects will reference the exact memory location, then change made by one will reflect those change in another object as well. This can lead to unpleasant side effects if the elements of values are changed via some other reference. Since we wanted to create an object replica, the Shallow copy will not fulfill this purpose.

Note: C++ compiler implicitly creates a copy constructor and assignment operator to perform shallow copy at compile time.

A shallow copy can be made by simply copying the reference.
Example:

```
class students(){
    int age;
    char * names;

    public:
    students(int age, char * names){
        this->age = age;

        // shallow copy
        this->names = names;
        // here we are putting the same array.
        // we are just copying the reference
    }
};
```

The above code shows shallow copying.

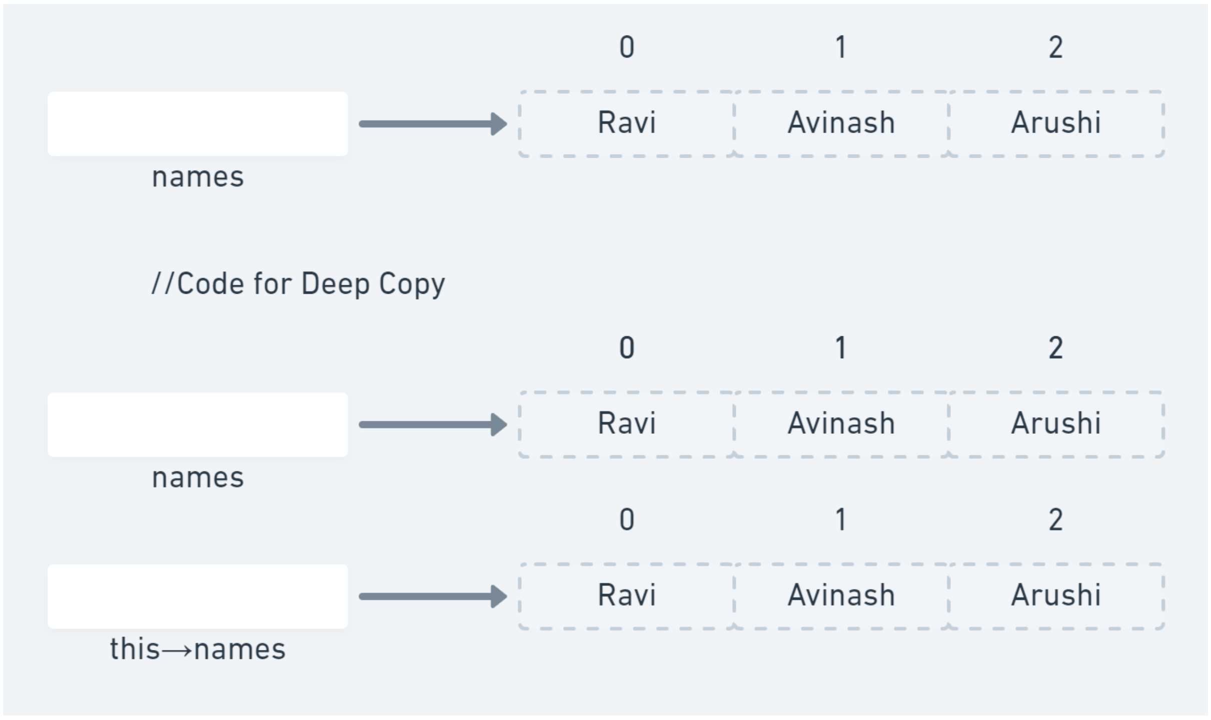


Deep Copy-

An object is created by copying all the fields, and it also allocates similar memory resources with the same value to the object. To perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is necessary to allocate memory to the other constructors' variables dynamically. A deep copy means creating a new array and copying over the values.
Changes to the array values referred to will not result in changes to the array data refers to.
Example:

```
class student(){
    int age;
    char * names;

    public:
    student(int age, char * names){
        this->age = age;

        //deep copy
        this->names = new char[strlen(names) + 1];
        strcopy(this->names, names);
        //Created new array and copied data
    }
};
```

The above code shows deep copying.



Interview Questions

1. **What is this pointer?**
   this pointer is accessible only inside the member functions of a class and points to the object which has called this member function.
2. **When is it necessary to use this pointer?**

   Suppose we have two local variables with the same name as the data members' names. Suppose you want to assign the local variable value to the data members. In that case, you won't be able to do until unless you use this pointer because the compiler won't know that you are referring to the object's data members unless you use this pointer.

3. **What is similar between deep copy and shallow copy?**

   Both are used to copy data between objects.

4. **What is the difference between deep copy and shallow copy?**

Shallow Copy	Deep Copy
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object.
Shallow Copy stores the copy of the original object and points the references to the objects.	Deep copy stores the copy of the original object and recursively copies the objects as well.
Shallow copy is faster.	Deep copy is comparatively slower.

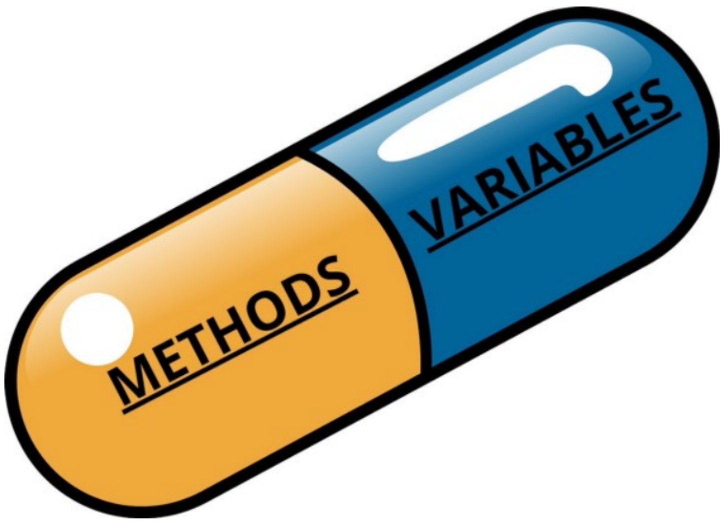
Encapsulation

Encapsulation-

Encapsulation refers to bundling data and the methods that operate on that data into a single unit. Many programming languages use encapsulation frequently in the form of classes. A class is an example of encapsulation in computer science in that it consists of data and methods that have been bundled into a single unit.

Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object. Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.

In other words: Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.



The general idea of this mechanism is simple. For example, you have an attribute that is not visible from the outside of an object. You bundle it with methods that provide read or write access. Encapsulation allows you to hide specific information and control access to the object's internal state.

Example:

```
#include <iostream>
using namespace std;
class Student {
    // private data members
    private:
    string studentName;
    int studentRollno;
    int studentAge;
    // get method for student name to access
    // private variable studentName
    public:
        string getStudentName() {
            return studentName;
        }
    // set method for student name to set
    // the value in private variable studentName
    void setStudentName(string studentName) {
        this -> studentName = studentName;
    }
    // get method for student rollno to access
    // private variable studentRollno
    int getStudentRollno() {
        return studentRollno;
    }
    // set method for student rollno to set
    // the value in private variable studentRollno
    void setStudentRollno(int studentRollno) {
        this -> studentRollno = studentRollno;
    }
    // get method for student age to access
    // private variable studentAge
    int getStudentAge() {
        return studentAge;
    }
    // set method for student age to set
    // the value in private variable studentAge
    void setStudentAge(int studentAge) {
        this -> studentAge = studentAge;
    }
};
int main() {
    Student obj;
    // setting the values of the variables
    obj.setStudentName("Avinash");
    obj.setStudentRollno(101);
    obj.setStudentAge(22);
    // printing the values of the variables
    cout << "Student Name : " << obj.getStudentName() << endl;
    cout << "Student Rollno : " << obj.getStudentRollno() << endl;
    cout << "Student Age : " << obj.getStudentAge();
}

Output:
Student Name : Avinash
Student Rollno : 101
Student Age : 22
```

Abstraction

Abstraction-

Abstraction means providing only some of the information to the user by hiding its internal implementation details. We just need to know about the methods of the objects that we need to call and the input parameters needed to trigger a specific operation, excluding the details of implementation and type of action performed to get the result.

Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

Real-life example: When you send an email to someone, you just click send, and you get the success message; what happens when you click send, how data is transmitted over the network to the recipient is hidden from you (because it is irrelevant to you).

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 members will be visible to the outside world and not. Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.

Example:

```cpp
#include <iostream>
using namespace std;
class abstraction {
    private:
        int a, b;
    public:
        // method to set values of private members
        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;
}
Output:
a = 10
b = 20
```

Advantages Of Abstraction

- Only you can make changes to your data or function, and no one else can.
- It makes the application secure by not allowing anyone else to see the background details.
- Increases the reusability of the code.
- Avoids duplication of your code.

Inheritance

Inheritance-

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).

The derived class inherits the features from the base class and can have additional features of its own.

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

Syntax:

```
class parent_class {
    //Body of parent class
};
class child_class: access_modifier parent_class {
    //Body of child class
};
```

Here, child_class is the name of the subclass, access_mode is the mode in which you want to inherit this sub-class, for example, public, private, etc., and parent_class is the name of the superclass from which you want to inherit the subclass.

Modes of Inheritance

1. **Public mode:** If we derive a subclass from a public base class. Then, the base class's public members will become public in the derived class, and protected class members will become protected in the derived class.

2. **Protected mode:** If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

3. **Private mode:** If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Base Class member Access Specifier	**Public**	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible	Not Accessible	Not Accessible

Example:
Suppose we have three classes with names: car, bicycle, and truck. The properties for each are as follows:

Car	**Bicycle**	**Truck**
→ Colour → Max Speed → Number of Gears	→ Colour → Max Speed → Is foldable?	→ Colour → Max Speed → Max weight

From above, we can see that two of the properties: Colour and MaxSpeed, are the same for every object. Hence, we can combine all these in one parent class and make three classes their subclass. This property is called Inheritance.

Technically, inheritance is defined as the process of acquiring the features and behaviors of a class by another class. Here, the class that contains these members is called the base class, and the class that inherits these members from the base class is called the derived class of that base class.

Code of the above example:

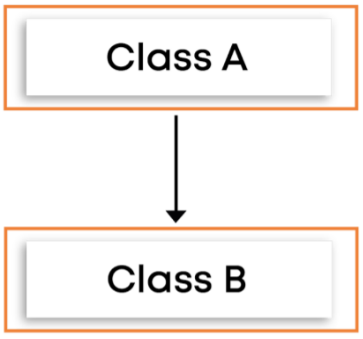
```
class vechile{
    public:
    string color;
    int max_speed;
};

class car : public vehicle{
    int num_gears;
};

class bicycle : public vehicle{
    bool is_foldable
};

class truck : public vehicle{
    int max_weight;
};
```

Types of Inheritance

Types of Inheritance-

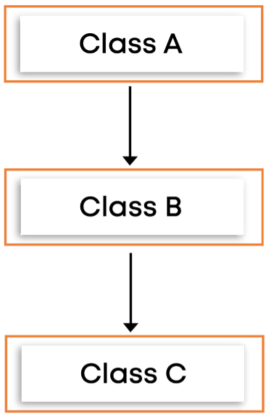
C++ supports five types of inheritance they are as follows:

1) **Single inheritance**
   In single inheritance, one class can extend the functionality of another class. There is only one parent class and one child class in single inheritances.



Single Inheritance

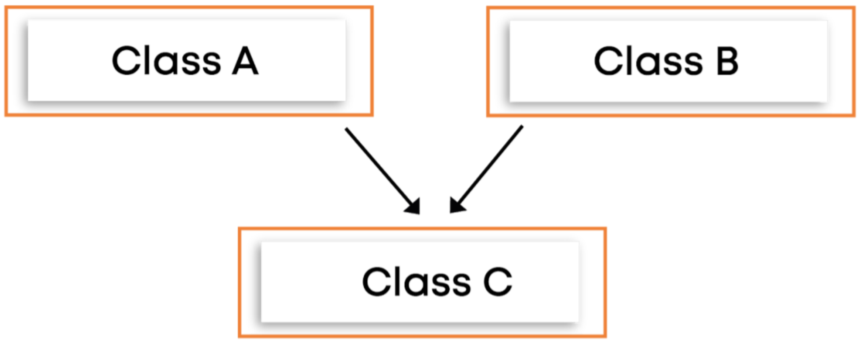
2) **Multilevel inheritance**
   When a class inherits from a derived class, and the derived class becomes the base class of the new class, it is called multilevel inheritance. In multilevel inheritance, there is more than one level.



Multi-Level Inheritance

3) **Multiple inheritance**

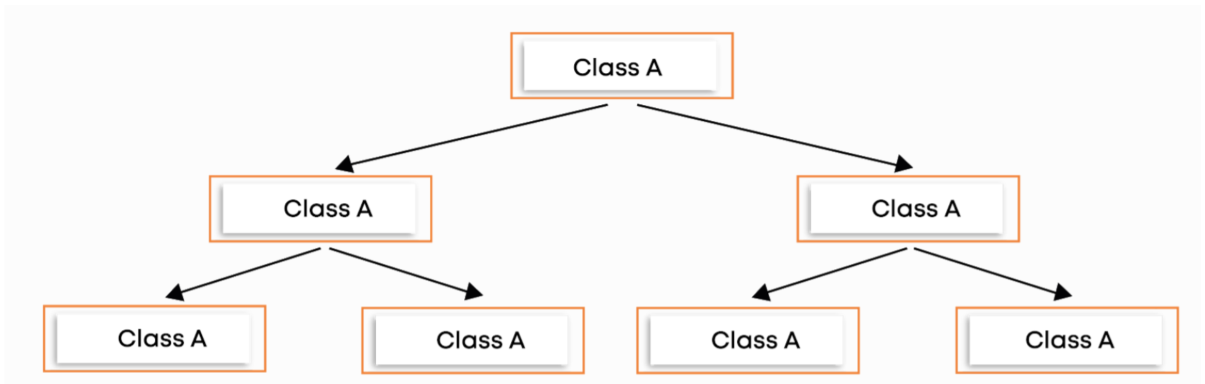
In multiple inheritance, a class can inherit more than one class. This means that a single child class can have multiple parent classes in this type of inheritance.



Multiple Inheritance

4) **Hierarchical inheritance**

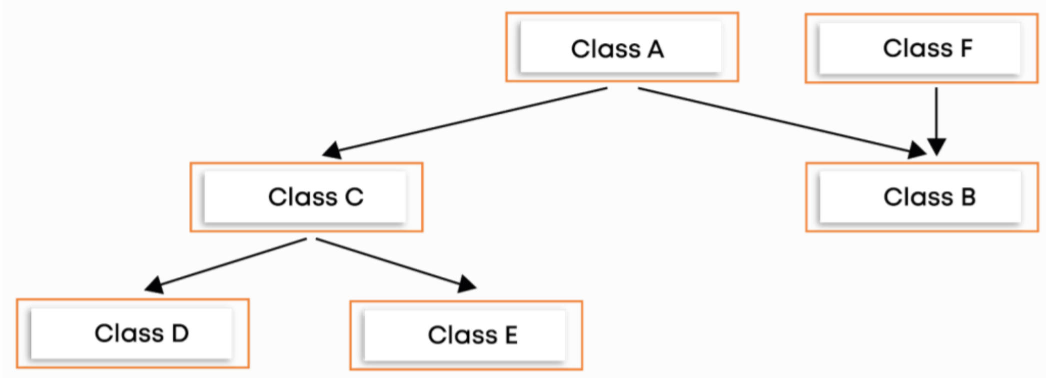
In hierarchical inheritance, one class is a base class for more than one derived class.



Hierarchical Inheritance

5) **Hybrid inheritance**

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.



Hybrid Inheritance- (a combination of Hierarchical and multiple)

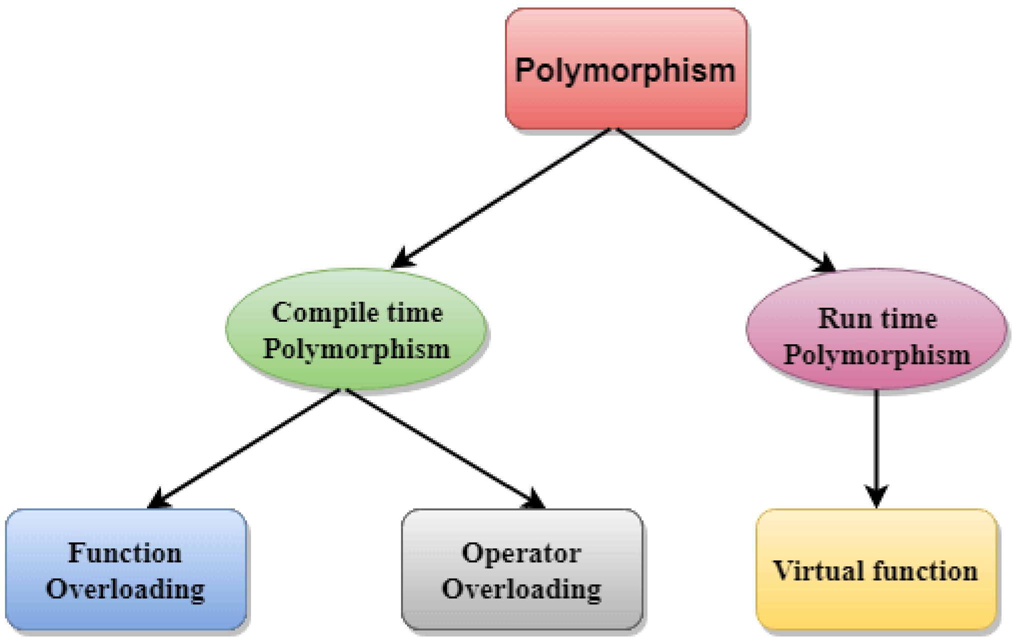
Polymorphism

Polymorphism-

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism is a concept that allows you to perform a single action in different ways. Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms. Let's understand polymorphism with a real-life example.

Real-life example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

There are two types of polymorphism in C++



❖ Compile Time Polymorphism:

Compile-time polymorphism is also known as static polymorphism. This type of polymorphism can be achieved through function overloading or operator overloading.

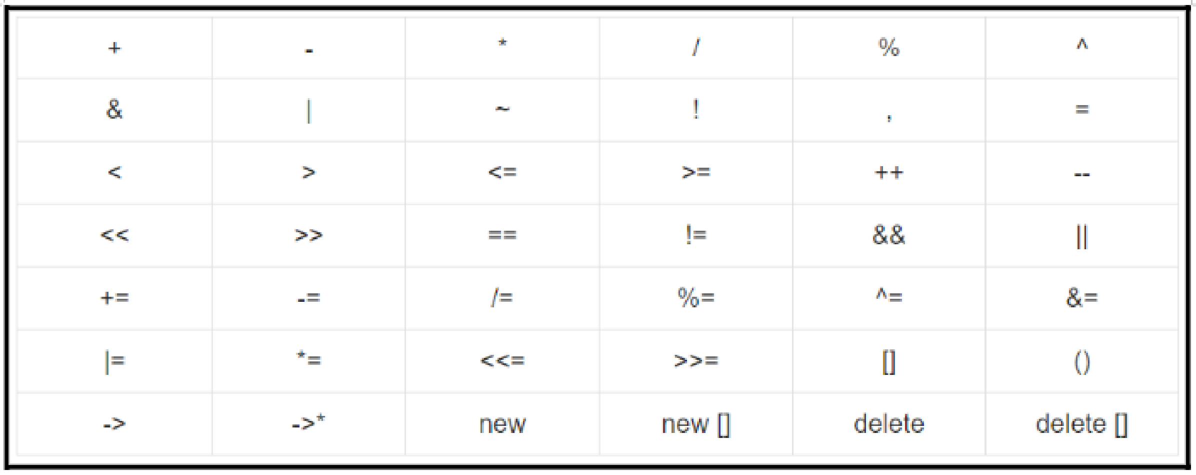
a) Function overloading:
When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of function overloading is that it increases the program's readability. Functions can be overloaded by using different numbers of arguments or by using different types of arguments. We have already discussed function overloading in detail in the previous module.

b) Operator Overloading:
C++ also provides options to overload operators. For example, we can make the operator ('+') for the string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. When placed between integer operands, a single operator, '+,' adds them and concatenates them when placed between string operands.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).
- Operators = and & are already overloaded in C++ to avoid overloading them.
- The precedence and associativity of operators remain intact.

List of operators that can be overloaded in C++:



+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

Example: Perform the addition of two imaginary or complex numbers.

```
#include<iostream>
using namespace std;
class Complex {
    private:
        int real, imag;
    public:
        Complex(int r = 0, int i = 0) {
            real = r;
            imag = i;
        }
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const & b) {
        Complex a;
        a.real = real + b.real;
        a.imag = imag + b.imag;
        return a;
    }
    void print() {
        cout << real << " + i" << imag << endl;
    }
};
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
Output:
12 + i9
```

❖ Runtime polymorphism:

Runtime polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymorphism.

Method overriding:
Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. In other words, whatever methods the parent class has by default are available in the child class. But, sometimes, a child class may not be satisfied with parent class method implementation. The child class is allowed to redefine that method based on its requirement. This process is called method overriding.

Rules for method overriding:

- The parent class method and the method of the child class must have the same name.
- The parent class method and the method of the child class must have the same parameters.
- It is possible through inheritance only.

Example:

```cpp
#include<iostream>
using namespace std;
class Parent {
    public:
        void show() {
            cout << "Inside parent class" << endl;
        }
};
class subclass1: public Parent {
    public: void show() {
        cout << "Inside subclass1" << endl;
    }
};
class subclass2: public Parent {
    public: void show() {
        cout << "Inside subclass2";
    }
};
int main() {
    subclass1 o1;
    subclass2 o2;
    o1.show();
    o2.show();
}
```

```
Output:
Inside subclass1
Inside subclass2
```

Interview Questions

1. **What is Encapsulation in C++? Why is it called Data hiding?**
The process of binding data and corresponding methods (behavior) into a single unit is called encapsulation in C++.
In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from accessing other classes. Thereby we can keep variables and methods safes from outside interference and misuse.
If a field is declared private in the class, it cannot be accessed by anyone outside the class and hides the fields. Therefore, Encapsulation is also called data hiding.

2. **What is the difference between Abstraction and Encapsulation?**

Abstraction	Encapsulation
Abstraction is the method of hiding unnecessary details from the necessary ones.	Encapsulation is the process of binding data members and methods of a program together to do a specific job without revealing unnecessary details.
Achieved through encapsulation.	You can implement encapsulation using Access Modifiers (Public, Protected & Private.)
Abstraction allows you to focus on what the object does instead of how it does it.	Encapsulation enables you to hide the code and data into a single unit to secure the data from the outside world.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.

3. **How much memory does a class occupy?**
Classes do not consume any memory. They are just a blueprint based on which objects are created. When objects are created, they initialize the class members and methods and therefore consume memory.

4. **Are there any limitations of Inheritance?**
   Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it also has limitations. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

5. **What is the difference between overloading and overriding?**
   Overloading is a compile-time polymorphism feature in which an entity has multiple implementations with the same name—for example, Method overloading and Operator overloading.
   Whereas Overriding is a runtime polymorphism feature in which an entity has the same name, but its implementation changes during execution. For example, Method overriding.

6. **What are the various types of inheritance?**
   The various types of inheritance include:
   - Single inheritance
   - Multiple inheritances
   - Multi-level inheritance
   - Hierarchical inheritance
   - Hybrid inheritance

7. **What are the advantages of Polymorphism?**
   There are the following advantages of polymorphism in C++:
   a. Using polymorphism, we can achieve flexibility in our code because we can perform various operations by using methods with the same names according to requirements.
   b. The main benefit of using polymorphism is when we can provide implementation to an abstract base class or an interface.
8. **What are the differences between Polymorphism and Inheritance in C++?**
   The differences between polymorphism and inheritance in C++ are as follows:

a. Inheritance represents the parent-child relationship between two classes. On the other hand, polymorphism takes advantage of that relationship to make the program more dynamic.

b. Inheritance helps in code reusability in child class by inheriting behavior from the parent class. On the other hand, polymorphism enables child class to redefine already defined behavior inside parent class.
Without polymorphism, a child class can’t execute its own behavior.

Virtual Function

Virtual Function-

A virtual function is a member function in the base class that we expect to redefine in derived classes. It is declared using the virtual keyword.
A virtual function is used in the base class to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to a derived class object.

Example:

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

class Base {
   public:
    virtual void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}

Output:
Derived Function
```

C++ determines which function is invoked at the runtime based on the type of object pointed by the base class pointer when the function is made virtual.

Pure Virtual Function:

A pure virtual function is a virtual function in C++ for which we need not write any function definition and only have to declare it. It is declared by assigning 0 in the declaration.
Syntax:

```
class A {
   public:
      virtual void s() = 0; // Pure Virtual Function
};
```

A pure virtual function (or abstract function) in C++ is a virtual function for which we can implement, But we must override that function in the derived class; otherwise, the derived class will also become an abstract class.

Abstract Class

Abstract Class-

Abstract classes can't be instantiated, i.e., we cannot create an object of this class. However, we can derive a class from it and instantiate the object of the derived class. An Abstract class has at least one pure virtual function.

Properties of the abstract classes:

- It can have normal functions and variables along with pure virtual functions.
- Prominently used for upcasting(converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as a base type), so its derived classes can use its interface.
- If an abstract class has a derived class, they must implement all pure virtual functions, or they will become abstract.

Example:

```
#include<iostream>
using namespace std;
class Base {
   public:
      virtual void s() = 0; // Pure Virtual Function
};

class Derived: public Base {
   public:
      void s() {
         cout << "Virtual Function in Derived_class";
      }
};

int main() {
   Base *b;
   Derived d_obj;
   b = &d_obj;
   b->s();
}
Output
Virtual Function in Derived_class
```

If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them and use their data members and member functions (except pure virtual functions).

Friend Function

Friend Function-

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

A class's friend function is defined outside that class's scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend function in C++ is a function that is preceded by the keyword "friend."

Syntax:

```
class class_name {
    friend data_type function_name(argument); // syntax of friend
function.
};
```

The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator.

Example:

```
#include <iostream>
using namespace std;
class Rectangle {
    private:
        int length;
    public:
        Rectangle() {
            length = 10;
        }
    friend int printLength(Rectangle); //friend function
};
int printLength(Rectangle b) {
    b.length += 10;
    return b.length;
}
int main() {
    Rectangle b;
    cout << "Length of Rectangle: " << printLength(b) << endl;
    return 0;
}
Output:
Length of Rectangle: 20
```

Characteristics of friend function:

- A friend function can be declared in the private or public section of the class.
- It can be called a normal function without using the object.
- A friend function is not in the scope of the class, of which it is a friend.
- A friend function is not invoked using the class object as it is not in the class's scope.
- A friend function cannot access the private and protected data members of the class directly. It needs to make use of a class object and then access the members using the dot operator.
- A friend function can be a global function or a member of another class.

Interview Questions

1. **Does every virtual function need to be always overridden?**
   No, It is not always mandatory to redefine a virtual function. It can be used as it is in the base class.

2. **What is an abstract class?**
   An abstract class is a class that has at least one pure virtual function in its definition. An abstract class can never be instanced (creating an object). It can only be inherited, and the methods could be overwritten.

3. **Can we have a constructor as Virtual?**
   Constructors cannot be virtual because they need to be defined in the class.

4. **What is a pure virtual function?**
   A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation. We only declare it. A pure virtual function is declared by assigning 0 in the declaration. See the following example.

5. **What are the characteristics of Friend Function?**
   - ★ A friend function is not in the scope of the class, in which it has been declared as friend.
   - ★ It cannot be called using the object of that class.
   - ★ It can be invoked like a normal function without any object.
   - ★ Unlike member functions, it cannot use the member names directly.
   - ★ It can be declared in public or private parts without affecting its meaning.
   - ★ Usually, it has objects as arguments.

6. **What is the output of this program?**

```
#include <iostream>
using namespace std;
class Box
{
    double width;
    public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};
void Box::setWidth( double wid )
{
    width = wid;
}
void printWidth( Box box )
{
    box.width = box.width * 2;
    cout << "Width of box : " << box.width << endl;
}
int main( )
{
    Box box;
    box.setWidth(10.0);
    printWidth( box );
    return 0;
}
```

Answer: 20

Explanation:

We are using the friend function for print width and multiplied the width value by 2, So we got the output as 20