OOPS

Part - 1

@kushagraraghav

Object Oriented Programming Concept in C++

- Topicwise explanation with example
- Interview question & answer

OOPs :-

OOP stands for Object-Oriented Programming.

Object-oriented programming is an approach to solving problems by using many concepts provided by OOP, such as inheritance, data binding, polymorphism, etc.

The primary goal of OOP is to connect the data and the functions that operate on it so that no other part of the code can access the data except that function.

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

• Why OOPs :-

Object-oriented programming has many advantages over procedural programming :-

- **Modularity:** OOP allows for the creation of reusable and modular code by breaking down complex systems into smaller, manageable objects.
- **Abstraction:** OOP allows developers to hide the implementation details of an object and only reveal a simplified, abstract view of it to the outside world.
- **Encapsulation:** OOP allows for the protection of an object's internal state by controlling access to its data and methods through the use of access modifiers.

- **Inheritance:** OOP allows for the creation of new objects based on existing ones, leading to a more efficient and organized codebase.
- **Polymorphism:** OOP allows for the use of a single interface to represent multiple types of objects, making the code more flexible and adaptable.
- **Better Organization**: OOP makes it easier to organize and structure the code, especially in large projects, and makes it more readable, maintainable and scalable.
- **Reusability:** OOP allows for the reuse of code through inheritance and polymorphism, leading to more efficient and cost-effective development.

• Class :-

A class is a user-defined data type that has data members and member functions, which can be accessed and used by creating an instance of that class.

A class is a group of similar objects. It can have fields, methods, constructors, etc.

```
class linkedIN //The class
{
 public: //Access Specifier
    int number;  // Data memeber
    string name; // Data memeber
};
```

• Object :-

An object is created from a class. It is a real-world entity that has data and a method or function.

When a class is defined, no memory is allocated, but when an object is created, memory is allocated. It is a runtime entity.

In OOP, everything is represented as an object, and when programmes are executed, the objects interact with each other by passing messages.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code.

It is sufficient to know the type of message accepted and the type of response returned by the objects.

Example :-

We have already created a class in the above example of a class, so now we can use this to create an object.

To create an object of LinkedIN class, specify the class name followed by the object name.

To access the class attributes, use the dot syntax (.) on the object:

```cpp
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:      //Access Specifier
    int number;  //Attribute
    string name;
};

int main()
{
    linkedIN obj1;          //Create an Object
    obj1.number = 10;       // Access Attribute & set value
    obj1.name = "kushagra";

    cout<<obj1.number<<endl;  //Print Attribute value
    cout<<obj1.name<<endl;   //Print Attribute value

    return 0;
}
```

Output :-

10
kushagra

- A single class creates multiple objects.

• class Methods :-

In c++, class Methods are functions that belong to the class. It is a function that operates on the data of an object of the class, and is defined within the class definition. Class methods can access and modify the member variables of an object, and can also be used to implement the behavior of an object.

There are two ways to define a member function:

1. Inside class definition
2. Outside class definition

1. Inside class defination :-

Member functions can be defined inside the class definition, right after the function declaration. This is known as an inline function definition. This approach is useful for small, simple functions that are called frequently, as it eliminates the overhead of a function call.

When defining a member function inside the class definition, the syntax is as follows:

```
class MyClass {
  public:
    void myFunction() { /* code */ }
};
```

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:      //Access Specifier

    void printThis()  //function define Inside the class
    {
        cout<<"Hello LinkedIN family"<<endl;
    }
};

int main()
{
    linkedIN obj1;          //Create an Object
    obj1.printThis();      //Call function

    return 0;
}
```

Output :-

Hello LinkedIN family

2. Outside class defination :-

To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed the **scope resolution(::)** operator, followed by the

name of the function. The scope resolution operator informs the compiler what class the member belongs to.

When defining a member function outside of the class definition, the syntax is as follows:

```
// class definition
class MyClass {
public:
    // member function declaration
    void myFunction();
};

// member function definition
void MyClass::myFunction() {
    // function body
}
```

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:      //Access Specifier

    void printThis();  //function declaration
};

void linkedIN :: printThis()
{
    cout<<"Hello LinkedIN family"<<endl; //function define Outside the class
}

int main()
{
    linkedIN obj1;          //Create an Object
    obj1.printThis();      //Call function

    return 0;
}
```

Output :-

Hello LinkedIN family

• Constructor :-

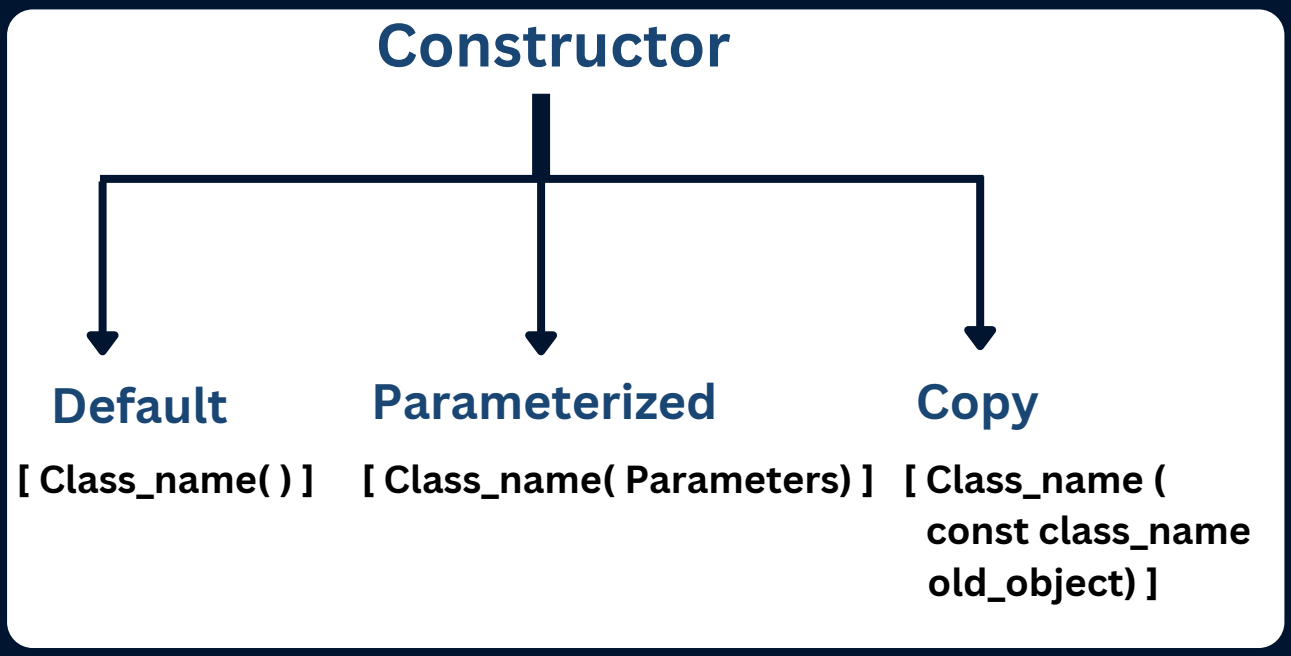
Constructor is a special method that is called automatically when an object is created.

Constructors have the same name as the class and may be defined inside or outside the class definition. The only restriction that applies to the constructor is that it must not have a return type or be void. It is because the constructor is automatically called by the compiler and is normally used to initialise values.

- Default Constructors don't have input arguments. However, copy and parameterized constructors have input arguments.
- If you do not openly provide a constructor of your own, then the compiler generates a default constructor for you.
- Constructors are also used at runtime to locate memory using the new operator.
- We can declare more than one constructor in a class, i.e., constructors can be overloaded.
- Constructor must be placed in public section of class.

- **Types of Constructor :-**

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor



1. Default Constructor :-

A constructor which has no argument and has the same name as the class is called the default constructor that is automatically generated by the compiler if no other constructors are defined for a class. It is also called a constructor with no parameters. It is invoked at the time of creating object.

The syntax of a default constructor in C++ is as follows:

```
class ClassName {
public:
    ClassName(); // default constructor
    // other members
};
```

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:
       linkedIN() //Default constructor
        {
         cout<<"Default constructor is invoked "<<endl;
        }
};

int main()
{
    linkedIN obj1;  //obj1 will call default constructor
    return 0;
}
```

Output :-

Default constructor is invoked

2. Parametrized Constructor :-

A parametrized constructor is a constructor that takes one or more arguments. These arguments are used to initialize the member variables of the class when an object of the class is created. Parameterized constructor is used to provide different values to distinct objects.

The syntax of a parametrized constructor in C++ is as follows:

```
class ClassName {
public:
    ClassName(type1 arg1, type2 arg2, ...); // parametrized constructor
    // other members
};
```

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:
     int number;
     string name;

       linkedIN(int a, string b) //Parameterized constructor
       {
        number = a;
        name   = b;

       }
       void display()
       {
        cout<<"Number = "<<number<<" & "<<" Name = "<<name<<endl;
       }
};

int main()
{
    linkedIN obj1 = linkedIN(22, "kushagra");  //obj1 will call Parameterized constructor
    linkedIN obj2 = linkedIN(23, "Dhruv");    //obj2 will call Parameterized constructor
                                              //with  different value
    obj1.display();      //print values
    obj2.display();
    return 0;
}
```

Output :-

Number = 22 & Name = kushagra
Number = 23 & Name = Dhruv

By using the scope-of-resolution operator, you can define a parameterized constructor outside the class.

3. Copy Constructor :-

In C++, a copy constructor is a special constructor that is used to create a copy of an object. It is called when an object is initialized with an existing object of the same type. The copy constructor typically takes a reference to the original object as its parameter and creates a new object that is a copy of the original. It is also used when an object is passed by value, as a function argument, or when it is returned by a function.

- The process of initialising members of an object through a copy constrictor is known as copy initialization.
- The copy constructor takes a reference to an object of the same class as an argument.
- The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

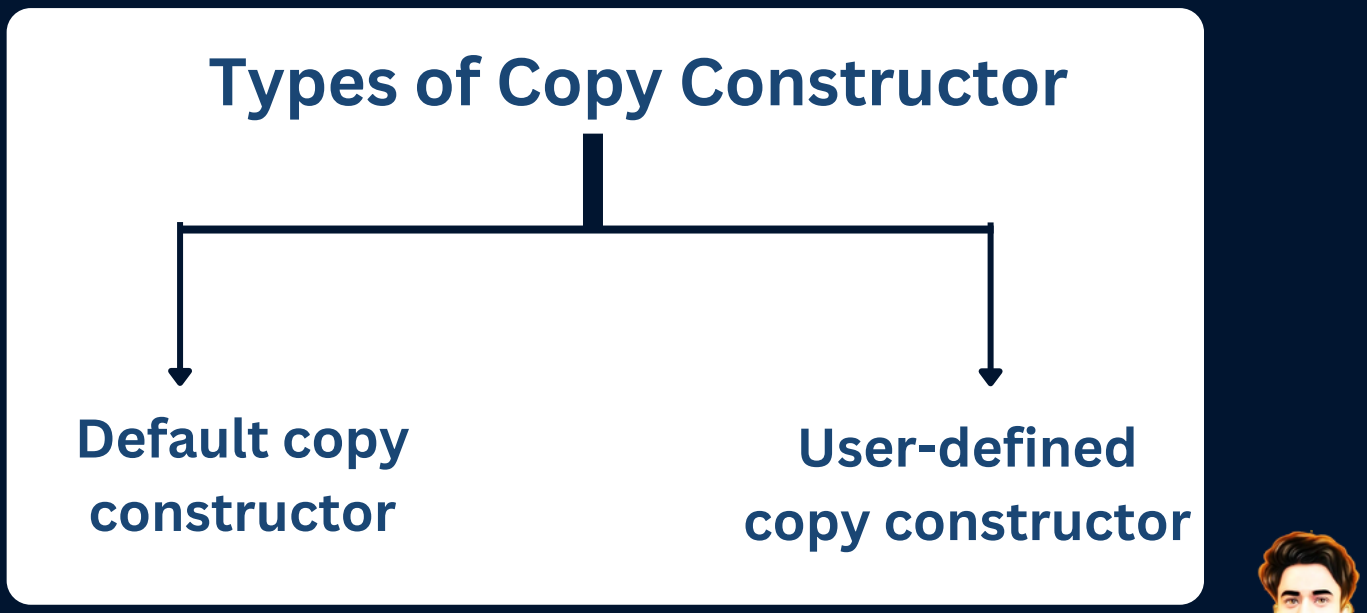
Example :-

```
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:
     int number;
     string name;

       linkedIN(int a, string b) //Parameterized constructor
       {
        number = a;
        name   = b;
```

```
    }
    linkedIN(linkedIN &obj1)   //copy constructor
    {
     number = obj1.number;
     name = obj1.name;
    }
    void display()
    {
     cout<<"Number = "<<number<<" & "<<" Name = "<<name<<endl;
    }
};

int main()
{
    linkedIN obj1(22, "kushagra");  //Calling the parameterized constructor
    linkedIN obj2 = obj1;           //Calling the copy constructor

    obj1.display();      //print values
    obj2.display();
    return 0;
}
```

Output :-

Number = 22 & Name = kushagra
Number = 22 & Name = kushagra



- **Types of copy Constructor :-**

1. Default copy Constructor
2. User-defined Constructor

1. Default copy-constructor :-

In C++, a default copy constructor is a constructor that is automatically generated by the compiler if the programmer does not explicitly define one. The default copy constructor performs a shallow copy of the object's members. This means that if the object contains pointers or references, only the memory addresses are copied, not the actual objects they point to. If the class has pointer or dynamic memory, it is recommended to define your own copy constructor that performs a deep copy.

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:
     int number;
     string name;

       linkedIN(int a, string b) //Parameterized constructor
       {
        number = a;
        name   = b;

       }

       void display()
       {
        cout<<"Number = "<<number<<" & "<<" Name = "<<name<<endl;
       }
};
```

```cpp
int main()
{
    linkedIN obj1(22, "kushagra");  //Calling the parameterized constructor
    linkedIN obj2 = obj1;           //Implicit Copy Constructor Calling

    obj1.display();      //print values
    obj2.display();
    return 0;
}
```

Output :-

Number = 22 & Name = kushagra
Number = 22 & Name = kushagra

2. User-defined Constructor :-

In C++, a user-defined copy constructor is a constructor that is explicitly defined by the programmer. This constructor is used to create a copy of an existing object, and it typically performs a deep copy of the object's members. A deep copy creates new memory for each object, rather than just copying the memory addresses as the default copy constructor does.

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class linkedIN    //The class
{
    public:
     int number;
     string name;

       linkedIN(int a, string b) //Parameterized constructor
       {
        number = a;
        name   = b;

       }
```

```
    linkedIN(linkedIN &obj1)   //copy constructor
    {
     number = obj1.number;
     name = obj1.name;
    }
    void display()
    {
     cout<<"Number = "<<number<<" & "<<" Name = "<<name<<endl;
    }
};

int main()
{
    linkedIN obj1(22, "kushagra");  //Calling the parameterized constructor
    linkedIN obj2 = obj1;           //Calling the copy constructor

    obj1.display();      //print values
    obj2.display();
    return 0;
}
```

Output :-

Number = 22 & Name = kushagra
Number = 22 & Name = kushagra

- **When the Copy Constructor function is invoked :-**

In the following cases, Copy Constructor is invoked:

- when a class object is passed by value as an argument.
- when we initialise the object with another object of the same class type that already exists.
- when the function returns an object of the same class type by value.

➤ when the compiler generates a temporary object.

• Copy elision :-

Copy elision is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it.

It is also known as copy omission.

when a temporary object is returned from a function and is immediately used to construct a new object, without the temporary object ever being explicitly named or accessed.

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class A
{
public:
    A(const char* str = "\0") //default constructor
    {
        cout <<"Default Constructor called" << endl;
    }

    A(const A &a) //copy constructor
    {
        cout <<"Copy constructor called" << endl;
    }
};
```

```
int main()
{
    A a1 = "copy me"; // Create object of class A
    return 0;
}
```

Output :-

Default Constructor called

- **Why isn't Copy Constructor called in the preceding example?**

According to theory, when the object "a1" is being constructed, one argument constructor is used to convert "copy me" to a temporary object, and that temporary object is copied to the object "a1." So the statement :-

A a1 = "copy me";

should be broken down by the compiler as follows:

A a1 = A("copy me");

- **When is a user-defined copy constructor needed ?**

If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource, like a file handle, a network connection, etc.

The default constructor makes only shallow copies. Deep copy is possible only with a user-defined copy constructor.

- **Shallow Copy :-**

A shallow copy is defined as the process of creating a copy of an object by copying the data of all the member variables as they are.

The C++ compiler implicitly creates a copy constructor and overloads the assignment operator in order to perform shallow copy at compile time.

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class rectangle {
private:
    int length;
    int breadth;
    int height;
public:
    void dimensions(int l, int b, int h)
    {
        length = l;
        breadth = b;
        height = h;
    }
    void display()
    {
        cout << " Length = " << length <<endl
             << " Breadth = " << breadth <<endl
             << " Height = " << height << endl;
    }
};

int main()
{
    rectangle R1, R3;

    R1.dimensions(4, 2, 6);
    R1.display();

    rectangle R2 = R1;   // When copying the data of an object at the time of initialization,
    R2.display();       //the copy is made through the copy constructor.

    R3 = R1;          //When copying the data of an object after initialization,
    R3.display();    // the copy is done through the Default Assignment Operator.

    return 0;
}
```

Output :-

Length = 4	Length = 4	Length = 4
Bredth = 2	Bredth = 2	Bredth = 2
Height = 6	Height = 6	Height = 6

- **Deep Copy :-**

Deep copy dynamically allocates the memory for the copy and then copies the actual value; both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class rectangle {
private:
    int length;
    int* breadth;
    int height;
public:
    rectangle()     //constructor
    {
        breadth = new int;
    }
    void dimensions(int l, int b, int h)    //function to get the dimension of rectangle
    {
        length = l;
        *breadth = b;
        height = h;
    }
    void display()     //display the dimension
    {
        cout << " Length = " << length <<endl
             << " Breadth = " << *breadth <<endl
             << " Height = " << height << endl;
    }
    rectangle(rectangle& dummy)    //parameterized constructor for implementing deep copy
    {
        length = dummy.length;
        breadth = new int;
        *breadth = *(dummy.breadth);
        height = dummy.height;
    }
    ~rectangle()          //destructor
    {
        delete breadth;
    }
};

int main()
{
    rectangle R1;

    R1.dimensions(4, 2, 6);   //set the dimension
    R1.display();

    rectangle R2 = R1;   // When copying the data of an object at the time of initialization,
    R2.display();        // all the resources will also get allocated to the new object

    return 0;
}
```

Output :-

Length = 4	Length = 4
Bredth = 2	Bredth = 2
Height = 6	Height = 6

- # Destructor :-

- Like constructor, destroyer is also a special member function. Destructors destroy the class objects created by constructors.

- It can be defined only once in a class. Like constructors, it is invoked automatically.

- Destructor has the same name as their class name, preceded by a tilde (~) symbol.

- The destructor is the only way to destroy the object created by the constructor. As a result, a destroyer cannot be overloaded.

- It is automatically called when an object goes out of scope.

- Destructor neither requires an argument nor returns any value.

- Destructors release the memory space occupied by the objects created by constructors.

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class LinkedIN
{
    public:
        LinkedIN()
        {
            cout<<"Constructor executed"<<endl;
        }

        ~LinkedIN()
            {
            cout<<"Destructor executed"<<endl;
            }
};

int main()
{
     LinkedIN l;

    return 0;
}
```

Output :-

Constructor executed
Destructor executed

• Access Specifiers:-

In C++, access specifiers are keywords used to set the accessibility of class members and methods. There are three main access specifiers :

1. Public
2. Private
3. Protected

1. Public :-

Members and Methods declared as public can be accessed from anywhere, both inside and outside of the class.

2. Private :-

Members and Methods declared as private can only be accessed from within the class. They are not accessible from outside the class.

3. Protected:-

Members and Methods declared as protected can be accessed from within the class and its derived classes. They are not accessible from outside the class.

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class LinkedIN {
public:
    int public_var;
    void public_method() {
        cout << "This is public method" << endl; }

private:
    int private_var;
    void private_method() {
        cout << "This is private method" << endl; }

protected:
    int protected_var;
    void protected_method() {
        cout << "This is protected method" << endl; }
};

int main() {
    LinkedIN obj;
    obj.public_var = 5;   // OK, public_var is public
    obj.public_method();  // OK, public_method is public
    obj.private_var = 10; // Error, private_var is private
    obj.private_method(); // Error, private_method is private
    obj.protected_var = 15; // Error, protected_var is protected
    obj.protected_method(); // Error, protected_method is protected
    return 0;
}
```

Output :-

This is public method

Because only the public method of the class is being called in the main function and it will print "This is public method" on the console.

However, the assignment to private_var and the call to private_method and protected_var and the call to protected_method will give compile errors because they are private and protected member and can only be accessed within the class.

• Encapsultaion :-

Encapsulation can be defined as the bundling of data and functions that work on that data within a single unit called a class.

In other words, encapsulation is the mechanism of hiding all the internal details of an object from the outside world. This makes it possible to change the implementation of an object without affecting the code that uses it. This is also known as data hiding and information hiding.

Encapsulation is used to improve code maintainability and reduce the risk of bugs

A real-life example of encapsulation would be a car. The car has several internal systems such as the engine, transmission, and fuel system. These systems are encapsulated within the car and operate independently of one another. The driver of the car only needs to know how to operate the controls (steering wheel, gas pedal, brakes) to drive the car, and does not need to know the specifics of how the engine, transmission, and fuel system work.

• Reasons why would we use encapsulation :-

1. **Data Hiding** : Encapsulation allows the developer to hide the implementation details of a class from other parts of the program. This makes it possible to change the implementation without affecting other parts of the code that use it.

2. **Abstraction** : Encapsulation provides a level of abstraction between the user and the implementation details of a class. This makes it easier to understand and use the class.

3. **Modularity** : Encapsulation allows for the creation of self-contained and modular code. This makes it easier to reuse and test the code.

4. **Security** : Encapsulation can be used to prevent unauthorized access to the data and functions of a class.

5. **Code Reusability** : Encapsulation allows the developer to change the internal implementation of a class without affecting other parts of the code that use it. This makes it easier to reuse the code.

Example :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class Car {
private:
    int speed;
    int gear;

public:
    void setSpeed(int s) {
        if(s >= 0)
            speed = s;
    }
    void setGear(int g) {
        gear = g;
    }
    int getSpeed() {
        return speed;
    }
    int getGear() {
        return gear;
    }
};
```

```
int main() {
    Car mycar;
    mycar.setSpeed(50);
    mycar.setGear(3);
    cout<<"Speed is : "<<mycar.getSpeed()<<endl;
    cout<<"Gear is : "<<mycar.getGear()<<endl;
    return 0;
}
```

Output :-

Speed is : 50
Gear is : 3

In above example, the data members "speed" and "gear" are private, meaning they can only be accessed within the class. The member functions "setSpeed", "setGear", "getSpeed", and "getGear" are public, meaning they can be called from outside the class to modify or access the private data members. This allows for the implementation of data validation and protection, as the private data members can only be changed through the public member functions, which can include checks to ensure valid input.

• Inheritance :-

In C++, Inheritance is a mechanism that allows a class to acquire all the properties and methods from another class. It's a way of creating a new class based on an existing class, while adding new features and functionality. This allows for code reuse, and a more organized and efficient object-oriented design.

Inheritance is defined using the ":" syntax, with the derived class listed before the ":" and the base class listed after. The derived class inherits all the properties and methods of the base class, and can also add new properties and methods of its own.

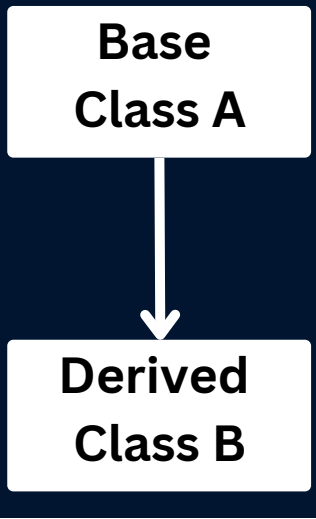
In real life, It's like creating a blueprint of a house, you can use the blueprint of a house to create a new house but with some changes or addition like a swimming pool or a garage. The new house is a derived class and the blueprint is the base class.

- C++ supports five types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multi-level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance :-

The inheritance in which a derived class is inherited from the only one base class.



In this, class 'A' is the base class, and class 'B' is the derived class.

- The syntax for single inheritance in C++ is as follows:

```
class DerivedClass : access_specifier BaseClass {
    // members of the derived class
};
```

Where **DerivedClass** is the name of the derived class, **access_specifier** is one of the access modifiers (public, protected, or private) and 'BaseClass' is the name of the base class.

Here is an example of single inheritance in C++ using a "Car" class that inherits from a "Vehicle" class:

In this example, the Car class inherits the members (i.e., the variables and methods) of the Vehicle class

Example :-

```
#include<bits/stdc++.h>
using namespace std;
class Vehicle {
  public:
    int wheels;
    void startEngine() {
        cout << "Engine started" << endl;
    }
};
```

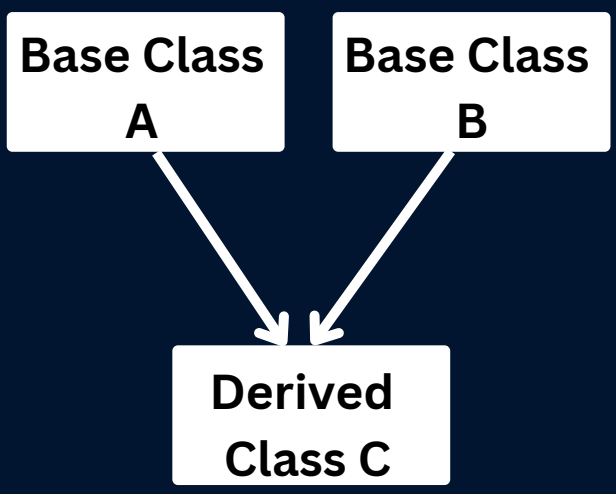
```cpp
class Car : public Vehicle {
public:
    int doors;
    void honkHorn() {
        cout << "Honk Honk" << endl;
    }
};

int main() {
    Car myCar;
    myCar.wheels = 4;
    myCar.doors = 4;
    myCar.startEngine();
    myCar.honkHorn();
    return 0;
}
```

Output :-

Engine Started
Honk Honk

2. Multiple Inheritance :-

Multiple Inheritance is a feature of C++ where a class can inherit from two or more than two class.



- The syntax for multiple inheritance in C++ is as follows:

```
class derived_class : access_specifier base_class1, access_specifier
base_class2, ...
{
    // members of derived class
};
```

- **derived_class** is the name of the class that is inheriting from multiple base classes.
- **access_specifier** is the access level (e.g., public, protected, or private) for the inheritance. The most commonly used access specifier for multiple inheritance is public, but you can also use protected or private.
- **base_class1, base_class2, ...** are the names of the classes that the derived class is inheriting from.

Example :-

Here is an example of multiple inheritance using a "RaceCar" class that inherits from both a "Car" class and a "RaceEngine" class:

```
#include<bits/stdc++.h>
using namespace std;
class Engine {
  public:
    int horsepower;
    void startEngine() {
        cout << "Engine started" << endl;
    }
};
```

```cpp
class Car {
public:
    int wheels;
    void honkHorn() {
        cout << "Honk Honk" << endl;
    }
};

class RaceEngine: public Engine {
public:
    void boost() {
        cout<<"Engine Boosted"<<endl;
    }
};

class RaceCar : public Car, public RaceEngine {
public:
    int nitrous;
    void activateNitrous() {
        cout<<"Nitrous activated"<<endl;
    }
};

int main() {
    RaceCar myRaceCar;
    myRaceCar.wheels = 4;
    myRaceCar.horsepower = 500;
    myRaceCar.nitrous = 100;
    myRaceCar.startEngine();
    myRaceCar.boost();
    myRaceCar.activateNitrous();
    myRaceCar.honkHorn();
    return 0;
}
```

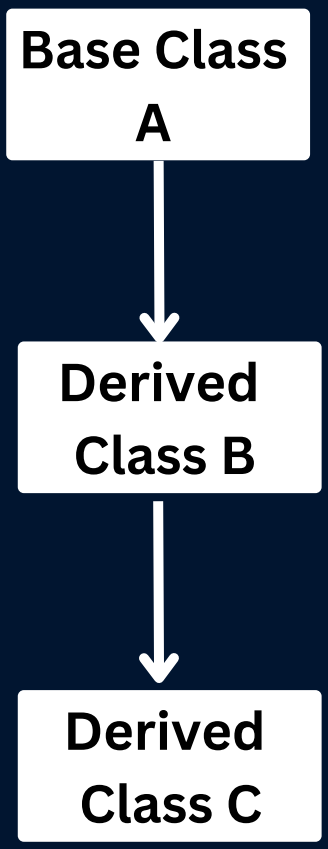
Output :-

Engine Started
Engine Boosted
Nitrous activated
Honk Honk

3. Multi-level Inheritance :-

In C++, multi-level inheritance is a feature that allows a class to inherit from another class, which in turn inherits from yet another class.

It's a chain of inheritance, where a derived class inherits from a base class, and the base class is derived from another base class.



The syntax for multi-level inheritance in C++ is very similar to the syntax for single inheritance. The main difference is that the derived class inherits from another derived class, which in turn inherits from a base class.

- The syntax for multi-level inheritance in c++ is as follows :-

```
class BaseClassA
{
    // body of the class A.
};
class DerivedClassB : public BaseClassA
{
    // body of class B.
};
class DerivedClassC : public DerivedClassB
{
    // body of class C.
};
```

Here's an example of how multi-level inheritance can be used to model a car using C++:

```
#include<bits/stdc++.h>
using namespace std;
class Vehicle {
  public:
    int wheels;
    string make;
    string model;

    void startEngine() {
        cout << "Engine started." << endl;
    }
};

class Car : public Vehicle {
  public:
    int doors;
    string color;

    void honkHorn() {
        cout << "Honk honk!" << endl;
    }
};
```

```
class SportsCar : public Car {
  public:
    int horsepower;
    int topSpeed;

    void accelerate() {
        cout << "Accelerating..." << endl;
    }
};

  int main() {
  SportsCar myCar;
  myCar.wheels = 4;
  myCar.make = "Ferrari";
  myCar.startEngine();
  myCar.honkHorn();
  myCar.accelerate();
  }
```

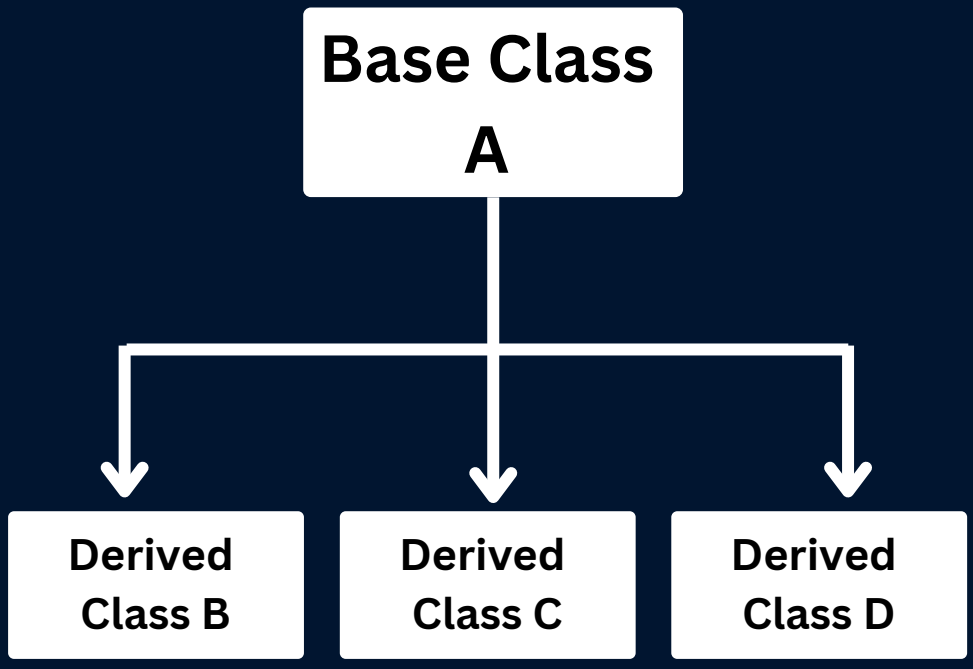
Output :-

Engine Started
Honk honk!
Accelerating. . .

This example shows how multi-level inheritance allows for modeling complex relationships and reusing code, making it easier to add new features to the class hierarchy.

4. Hierarchical Inheritance :-

In C++, hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Hierarchical inheritance allows for code reuse and helps to organize the class hierarchy in a logical and organized way.

It allows for modeling complex relationships and reusing code, making it easier to add new features to the class hierarchy.

The syntax for Hierarchical inheritance in C++ is as follows:

```
class BaseClassA
{
    // body of the class A.
};
class DerivedClassB : public BaseClassA
{
    // body of class B.
};
class DerivedClassC : public BaseClassA
{
    // body of class C.
};
class DerivedClassD : public BaseClassA
{
    // body of class D.
};
```

Here is an example of hierarchical inheritance in C++ where a class "Car" serves as the base class and "SUV" and "Sedan" classes inherit from it:

```cpp
#include<bits/stdc++.h>
using namespace std;
class Car {
  protected:
    int wheels;
    int seats;
  public:
    Car(int w, int s) {
      wheels = w;
      seats = s;
    }
    void display() {
      cout << "Wheels: " << wheels << endl;
      cout << "Seats: " << seats << endl;
    }
};

class SUV: public Car {
  public:
    SUV(int w, int s): Car(w, s) {}
};

class Sedan: public Car {
  public:
    Sedan(int w, int s): Car(w, s) {}
};

int main() {
    SUV suv(4, 7);
    Sedan sedan(4, 5);
    suv.display();
    sedan.display();
    return 0;
}
```

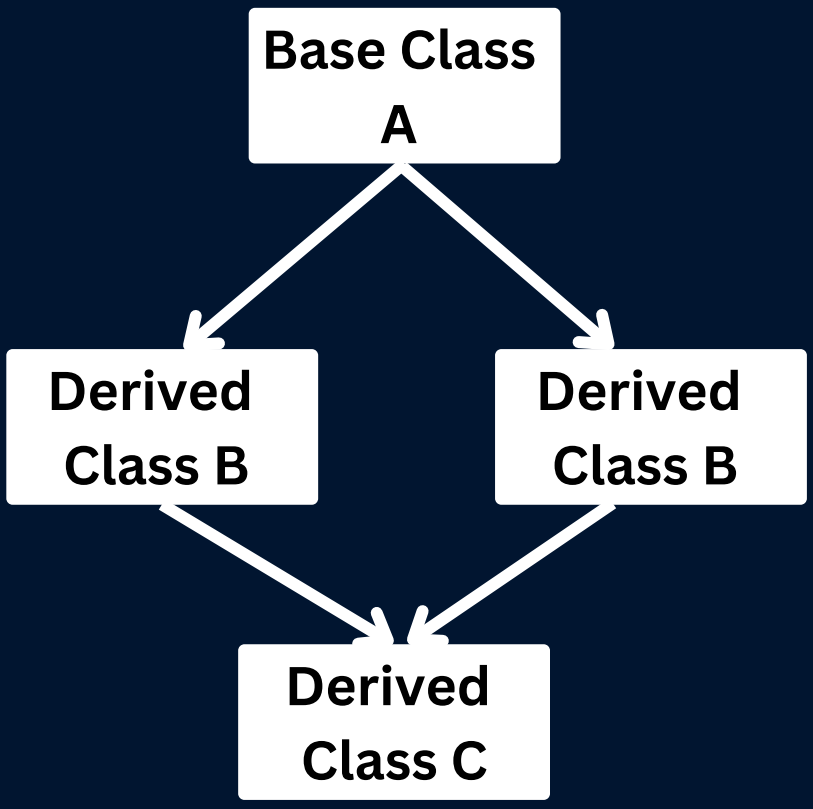
Output :-

Wheels : 4
Seats : 7
Wheels : 4
Seats : 5

Here, the class "SUV" and "Sedan" inherit from the class "Car" and can access its protected member variables and methods. The constructor for each derived class calls the constructor for the base class to initialize its member variables. In main method, we can create object for SUV and Sedan and can access the display method which is inherited from Car class.

5. Hybrid Inheritance :-

In C++, Hybrid inheritance is a combination of multiple types of inheritance, typically a combination of both single and multiple inheritance.



- The syntax for Hybrid inheritance in C++ is as follows:

```
class BaseClassA
{
    // body of class A.
};
class DerivedClassB : public BaseClassA
{
    // body of class B.
};
```

```
class DerivedClassC : public BaseClassA
{
    // body of class C.
};
class DerivedClassD : public DerivedClassB :  public DerivedClassC
{
    // body of class D.
};
```

Here is an example of hybrid inheritance in C++ where a class "Car" serves as the base class, "SUV" class inherits from "Car" and "DieselEngine" class, and "Sedan" class inherits from "Car" and "PetrolEngine" class:

```
#include<bits/stdc++.h>
using namespace std;
class Car {
  protected:
    int wheels;
    int seats;
  public:
    Car(int w, int s) {
      wheels = w;
      seats = s;
    }
    void display() {
      cout << "Wheels: " << wheels << endl;
      cout << "Seats: " << seats << endl;
    }
};

class Engine {
  protected:
    string fuelType;
  public:
    Engine(string ft) {
      fuelType = ft;
    }
    void displayEngine() {
      cout << "Fuel type: " << fuelType << endl;
    }
};

class SUV: public Car, public Engine {
  public:
    SUV(int w, int s, string ft): Car(w, s), Engine(ft) {}
};
```

```
class Sedan: public Car, public Engine {
  public:
    Sedan(int w, int s, string ft): Car(w, s), Engine(ft) {}
};

int main() {
    SUV suv(4, 7, "Diesel");
    Sedan sedan(4, 5, "Petrol");
    suv.display();
    suv.displayEngine();
    sedan.display();
    sedan.displayEngine();
    return 0;
}
```

Output :-

Wheels : 4
Seats : 7
Fuel type : Diesel
Wheels : 4
Seats : 5
Fuel type : Petrol

Here the class "SUV" inherits from both "Car" and "DieselEngine" classes and the class "Sedan" inherits from both "Car" and "PetrolEngine" classes. That's how we are combining different type of inheritance to achieve Hybrid Inheritance.

- **Polymorphism :-**

Generally, The word “polymorphism” means having many forms.

In C++, polymorphism refers to the ability of a function or an operator to behave in different ways depending on the context in which it is called. This allows for a single function or operator to work with multiple types of data or objects.

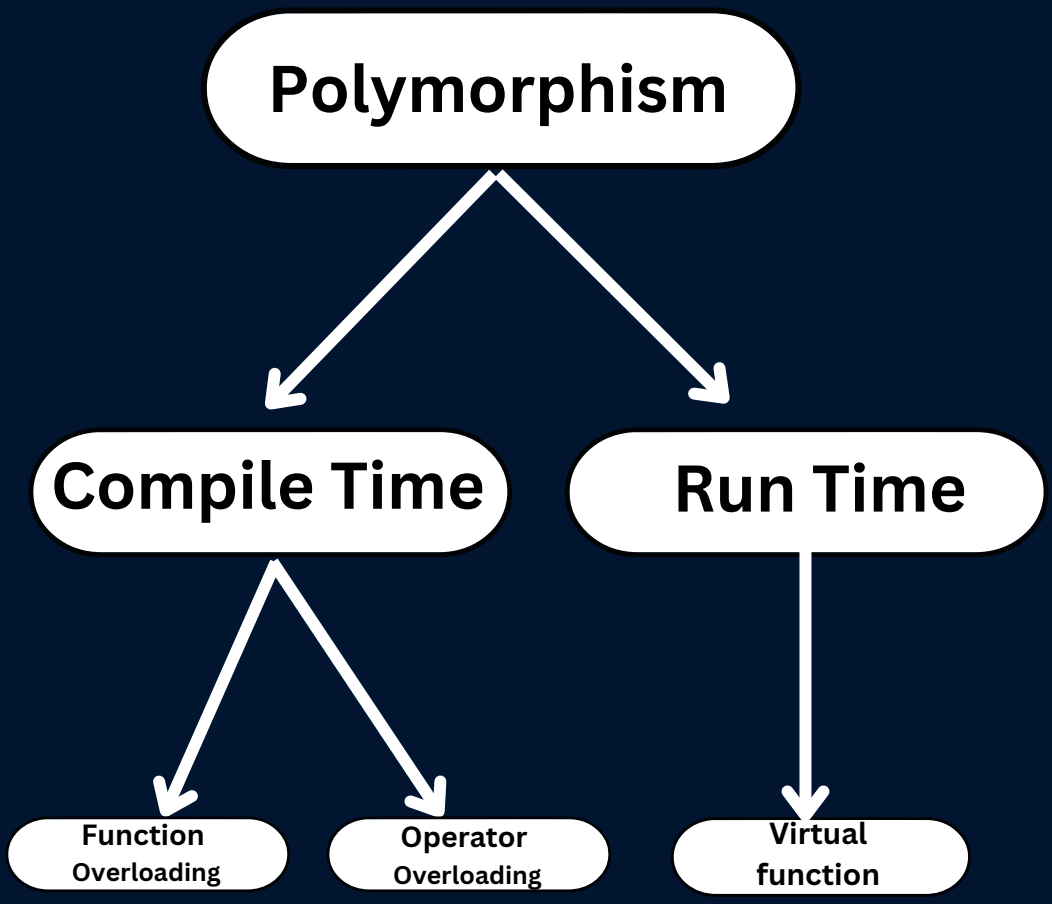
This is achieved through the use of virtual functions and function overloading.

This allows for more flexibility and reusability in code, as the same function or operator can be used in different situations without needing to be rewritten.

For example, imagine you have a function that is designed to draw a shape on a screen. Without polymorphism, you would need to write separate functions for each type of shape (e.g. drawCircle, drawSquare, drawTriangle). With polymorphism, you can write a single function called "drawShape" that can take in any type of shape and draw it on the screen.

- Types of Polymorphism :-

1. Compile-time Polymorphism
2. Run-time Polymorphism



1. Compile-time Polymorphism :-

Compile-time polymorphism, also known as "static polymorphism", is achieved through function overloading and operator overloading.

1.1 Function Overloading :-

Function overloading is a feature that allows multiple functions to have the same name but different parameter lists.

This means that multiple functions with the same name can be defined in a single scope, and the correct function to call will be determined by the number, types, and order of the arguments passed to the function.

Function overloading is a compile-time polymorphism. The correct function to call is determined by the compiler at the time of compilation based on the types of the arguments passed to the function.

Here is an example of function overloading in C++:

Program of function overloading with different types of arguments.

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

void print(int x) {
    cout << "Integer: " << x << endl;
}

void print(double x) {
    cout << "Double: " << x << endl;
}

void print(string x) {
    cout << "String: " << x << endl;
}

int main() {
    int i = 5;
    double d = 3.14;
    string s = "Hello";

    print(i); // calls the first version of print(int x)
    print(d); // calls the second version of print(double x)
    print(s); // calls the third version of print(string x)
    return 0;
}
```

Output :-

Integer : 5
Double : 3.14
String : Hello

In this example, the function "print" is overloaded three times, each time taking a different type of argument (int, double, string). When the function is called in the

main function with an int, a double and a string, the correct version of the function is called based on the type of the argument passed to it.

Program of function overloading when number of arguments vary.

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

void print(int x, int y) {
  cout << "Printing integers: " << x << " and " << y << endl;
}

void print(int x) {
  cout << "Printing integer: " << x << endl;
}

int main() {
  print(5);
  print(5,6);
  return 0;
}
```

Output :-

Printing Integer : 5
Printing Integers : 5 and 6

1.2 Operators Overloading :-

Operator overloading is a feature in C++ that allows you to customize the way operators (like +, -, *, /, etc) work with objects of a user-defined class. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading

Also, it's not all operator can be overloaded, only a certain set of operators are available for overloading.

For example, if you have a class for complex numbers, you can overload the + operator so that it can add two complex numbers together in a way that makes sense for that class. Instead of having to write code like c3 = add_complex(c1, c2), you can simply write c3 = c1 + c2.

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

class Complex {
  public:
    double real, imag;
    Complex(double real = 0, double imag = 0) : real(real), imag(imag) {}
    Complex operator+(const Complex &c) {
      return Complex(real + c.real, imag + c.imag);
    }
};

int main() {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex c3 = c1 + c2;
    cout << c3.real << " + " << c3.imag << "i" << endl;
    return 0;
}
```

```
int main() {
    Complex c1(1, 2);
    Complex c2(3, 4);
    Complex c3 = c1 + c2;
    cout << c3.real << " + " << c3.imag << "i" << endl;
    return 0;
}
```

Output :-

4 + 6i

- ## Operator that can be overloaded :-

1. Binary Arithmetic -> +, -, *, /, %
2. Unary Arithmetic -> +, -, ++, —
3. Assignment -> =, +=,*=, /=,-=, %=
4. Bit- wise -> & , | , << , >> , ~ , ^
5. De-referencing -> (->)
6. Dynamic memory allocation and De- allocation -> New, delete
7. Subscript -> [ ]
8. Function call -> ()
9. Logical -> &, | |, !
10. Relational -> >, < , = =, <=, >=

- ## Operator that can't be overloaded :-

1. sizeof
2. typeid
3. Scope resolution (::)
4. Class member access operators (.(dot), .* (pointer to member operator))
5. Ternary or conditional (?:)

- **Rules for Operator Overloading :-**

1. Operators can only be overloaded as member functions or non-member functions. They cannot be overloaded as friend functions.
2. The overloaded operator should perform a function that is consistent with its usual meaning. For example, when overloading the + operator for a class representing complex numbers, it should return a new object representing the sum of the two complex numbers.
3. When overloading binary operators, the left-hand operand should always be of the class type, whereas the right-hand operand can be of any type.
4. It is not possible to change the precedence or associativity of operators when overloading them.
5. The keyword operator must be used to indicate that the function is an operator overload.
6. The operator overload function should have the same number of arguments as the original operator.
7. Overloading operators should improve the readability and expressiveness of your code, not make it more difficult to understand.
8. The = operator should always return a reference to the object on the left-hand side.
9. Overloading the = operator should also overload the copy constructor and destructor.

2. Run-time Polymorphism :-

Run-time polymorphism, also known as dynamic polymorphism or late binding, is a feature of C++ that allows a function or method to be called on an object of a derived class through a pointer or reference of its base class. This allows for the creation of more general and flexible code, as the specific type of the object being operated on is not known until runtime.

In C++, run-time polymorphism is achieved through virtual functions

2.1 Operators Overriding :-

In C++, Function overriding is a feature that allows a derived class to provide a different implementation for a virtual function that is already present in its base class. The process of providing a new implementation for an inherited function is known as "overriding" the function.

Here is an example of function overriding in C++:

```cpp
#include <iostream>
using namespace std;
class Shape {
public:
  virtual void draw() {
    cout << "Drawing a shape" << endl;
  }
};

class Circle : public Shape {
public:
  void draw() {
```

```
    cout << "Drawing a circle" << endl;
  }
};

int main() {
  Shape* shape = new Circle();
  shape->draw(); // Output: Drawing a circle
  return 0;
}
```

Output :-

Drawing a circle

This is because, in the main function, we create a pointer of Shape class **'Shape* shape = new Circle(); '** which actually points to a Circle object. When we call shape->draw() it calls the overridden method of Circle class which is 'void draw() { cout << "Drawing a circle" << endl; }' and it prints "Drawing a Circle"

- **Virtual function :-**

In C++, virtual functions are member functions of a class that can be overridden by derived classes. They are declared with the keyword **virtual** and have the same function signature in both the base class and the derived class.

It is worth noting that a virtual function in C++ can also be pure virtual function which is declared by virtual return_type function_name() = 0; and that class is called as Abstract class and cannot be instantiated.

Example same as Above function overriding example

• Abstraction :-

In C++, abstraction is the process of hiding the implementation details of an object and exposing only the necessary information to the user. This allows the user to interact with the object without needing to know how it works internally.

Abstraction in C++ can be achieved through the use of abstract classes, interfaces, and encapsulation.

- **Types of Abstraction :-**

1. Data Abstraction.
2. Functional Abstraction

Another type of abstraction is Object-Oriented Abstraction, it is a way of designing software systems by breaking them down into objects that interact with one another.

1. Data Abstraction :-

In C++, data abstraction is the process of hiding the implementation details of data and exposing only the necessary information to the user. This allows the user to interact with the data without needing to know how it is stored or represented internally.

A real-life example of data abstraction could be a remote control for a TV, it provides a user interface with buttons for

power, volume, channel, etc. The user can interact with the TV without needing to know how the TV works internally, how the signals are processed, or how the data is stored and represented internally.

- **Data Abstraction can be achieved in two ways :-**

1. Abstraction using classes
2. Abstraction in header files.

1. Abstraction using class :-

In C++, abstraction can be achieved using classes by creating an abstract base class that defines an interface and leaves the implementation details to be defined by derived classes.

An abstract class is a class that cannot be instantiated and is used as a base class for other classes.

Here's an example of abstraction using a class in C++:

```
#include <iostream>
using namespace std;
class Shape {
public:
  virtual double area() = 0;
};

class Rectangle : public Shape {
private:
  double width, height;
public:
  Rectangle(double width, double height) {
    this->width = width;
    this->height = height;
  }
  double area() {
    return width * height;
  }
};
```

```cpp
class Circle : public Shape {
private:
  double radius;
public:
  Circle(double radius) {
    this->radius = radius;
  }
  double area() {
    return 3.14159 * radius * radius;
  }
};

int main() {
  Shape* shape = new Rectangle(5, 10);
  cout << "Area of the rectangle: " << shape->area() << endl;
  shape = new Circle(5);
  cout << "Area of the circle: " << shape->area() << endl;
  return 0;
}
```

Output :-

Area of the rectangle : 50
Area of the circle : 78.537

In this example, the Shape class is an abstract class that defines an interface with the pure virtual function area(). The Rectangle and Circle classes both inherit from Shape and provide an implementation for the area() function. By using an abstract class, the implementation details of how to calculate the area of a shape are hidden from the user, and the user can interact with the object without needing to know how it works internally.

2. Abstraction in header files :-

Abstraction in header files is a programming technique that allows you to define a common interface for different classes or functions, without exposing the implementation details. This can be achieved by separating the interface and implementation of a class or function into separate header and source files.

1. **Header files**: The header file contains the declarations of variables, functions, and classes that can be used in other parts of the program. By using abstract classes, interfaces, pure virtual functions, and templates in header files, you can create a clear and consistent interface for other parts of the program to use, without exposing the implementation details of those classes and data types.
2. **Source files:** The source file contains the implementation of the functions and classes declared in the header file. This allows you to separate the interface and implementation of a class or function, making it easier to understand and maintain the code.

Here is a example of abstraction in header files in C++:

```
#ifndef MYLIBRARY_H
#define MYLIBRARY_H

class MyAbstractClass {
  public:
    virtual void displayData() = 0;
    virtual void processData() = 0;
};

#endif
```

This is the header file, in it we have defined an abstract class **MyAbstractClass** with two pure virtual functions **displayData()** and **processData()**. These functions are just the declaration and don't have any implementation.

```cpp
#include <iostream>
#include "MyLibrary.h"
using namespace std;
class MyDerivedClassA: public MyAbstractClass {
  private:
    int data;
  public:
    MyDerivedClassA(int d) {
      data = d;
    }
    void displayData() {
      cout << "Data: " << data << endl;
    }
    void processData() {
      data *= 2;
    }
};

int main() {
    MyDerivedClassA a(5);
    a.displayData(); // will display "Data: 5"
    a.processData();
    a.displayData(); // will display "Data: 10"
    return 0;
}
```

Output :-

Data : 5
Data : 10

This is the source file, in which we have defined a class MyDerivedClassA that inherits from the MyAbstractClass and implements the two pure virtual functions displayData().

• this pointer :-

In C++, the **this** pointer is a pointer that points to the current object of a class. It is passed as an implicit argument to all non-static member functions and it can be used to access the members of the object within the member function.

The **this** pointer is automatically passed to the member function when it is called, and it can be used to access the data members and other member functions of the class. The **this** pointer is also useful for resolving ambiguity when a member function has a parameter with the same name as a data member of the class.

Here is an example of how the **this** pointer can be used in a C++ class:

```cpp
#include <iostream>
using namespace std;
class MyClass {
    private:
        int data;
    public:
        MyClass(int d) {
            data = d;
        }
        void displayData() {
            cout << "Data: " << this->data << endl;
        }
};

int main() {
    MyClass myObject(5); // Creates an object of MyClass and initializes the data member with 5
    myObject.displayData(); // Outputs "Data: 5"
    return 0;
}
```

Output :-

Data : 5

In this example, the main function creates an object of the MyClass class, passing the value 5 as an argument to the constructor. This sets the value of the data member of the object to 5. Then, the main function calls the displayData() member function on the object, which uses the this pointer to access the data member and outputs its value to the console.

• static :-

In C++, the "static" keyword is used to indicate that a variable or function has a single, shared instance for the entire program, rather than a separate instance for each object of a class.

There are a few main ways that the "static" keyword can be used in C++:

1. **Static member variables:** A static member variable is a variable that is shared by all objects of a class. It is declared with the "static" keyword and is typically initialized outside the class. Only one copy of the variable exists, regardless of how many objects of the class are created.
2. **Static member functions:** A static member function is a function that can be called on the class itself, rather than on an object of the class. It can only access static member variables, and cannot access non-static member variables or call non-static member functions.

3. **Static local variables:** A static local variable is a variable that is created only once and retains its value between .....function calls.
4. **Global variables:** A global variable can be declared as static , so that it can only be accessed within the file in which it is defined.
5. **Namespace:** You can define the static variable inside a namespace and make sure that it is only accessible within that namespace.

Here is an example of the "static" keyword in C++:

```cpp
#include <iostream>
using namespace std;
class Example {
    public:
        static int x;
        static void setX(int value) {
            x = value;
        }
};

int Example::x = 0;  //static variables should be defined outside the class

int main() {
    Example obj1;
    Example obj2;
    obj1.setX(5);
    obj2.setX(10);
    cout << obj1.x <<endl; // Outputs "10"
    cout << obj2.x <<endl; // Outputs "10"
    return 0;
}
```

Output :-

10
10

In this example, we have a class called "Example" with a static member variable "x" and a static member function "setX". The static member variable "x" is shared by all objects of the class, meaning that any changes made to it will be visible to all objects. In the main function, we create two objects of the class "Example" and call the "setX" function on both of them. The value of x is changed to 10 for both objects, and when we access x variable using the objects, the output is 10 for both.

It's important to note that static member functions can only access static member variables and cannot access non-static member variables or call non-static member functions.

- **Enumeration :-**

In C++, an enumeration (or "enum") is a way to define a set of named integer constants. Enumerations are defined using the "enum" keyword, followed by a list of enumerators (the named constants) separated by commas.

If we assign a float value in a character value, then the compiler generates an error. In the same way if we try to assign any other value to the enumerated data types, the compiler generates an error. Enumerator types of values are also known as enumerators. It is also assigned by zero the same as the array. It can also be used with switch statements.

Here is an example of how to define and use an enumeration in C++:

```cpp
#include <iostream>
using namespace std;
enum year {
    January,
    February,
    March,
    April,
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December
};

int main()
{
    int i;

    for (i = January; i <= December; i++)
        cout << i << " ";

    return 0;
}
```

Output :-

0 1 2 3 4 5 6 7 8 9 10 11

- # Friend Function :-

In C++, a friend function is a non-member function that has been given the permission to access the private and protected members of a class. Friend functions are declared with the "friend" keyword, followed by the function declaration.

The syntax for declaring a friend function in C++ is as follows:

```
class ClassName {
    private:
        //private members
    protected:
        //protected members
    public:
        //public members
    friend return_type function_name(parameter_list);
};
```

The keyword "friend" is used to indicate that the function is a friend of the class.

It's also possible to declare a friend function inside the class definition, but define it outside, like this:

```
class ClassName {
    private:
        //private members
    protected:
        //protected members
    public:
        //public members
    friend return_type function_name(parameter_list);
};

return_type ClassName::function_name(parameter_list) {
    // function body
}
```

It's important to note that friend functions are not a part of the class, they are just given permission to access the class's private and protected members, but they do not have access to the "this" pointer and cannot be called using the dot operator.

Here is an example of a friend function in C++:

```cpp
#include <iostream>
using namespace std;
class MyClass {
private:
    int x;

public:
    MyClass(int val) {
    x = val;
    }
    friend int add(MyClass obj);
};

int add(MyClass obj) {
    return obj.x + 5; // can access private variable x
}

int main() {
    MyClass obj(10);
    cout << add(obj); // prints 15
    return 0;
}
```

Output :-

15

In this example, the function "add" is declared as a friend of the class "MyClass". The function can access the private variable "x" of any object of the class "MyClass", even though the function itself is not a member of the class.

Features of Friend Function:

- A friend function is a non-member function that has been granted access to the private and protected members of a class.
- Friend function is declared with keyword friend.
- A friend function is not in the scope of the class to which it has been granted access, so it cannot be called using the dot operator (.) or the scope resolution operator (::).
- Friend function is not a member function of the class.
- Friend function can be a member function of another class.
- Friend function can access the private and protected members of the class in which it is declared as a friend.
- Friend function can be declared either in public, protected or private section.
- Friend function can not be called using the object of the class and it should be called directly.
- Friend functions are typically used to implement functions that need access to the internal state of an object, but do not need to be part of the class interface.

• Friend class :-

In C++, a friend class is a class that has been granted access to the private and protected members of another class. This is done by including a "friend" keyword in the class definition, followed by the class name.

Here's an example of how you might use a friend class in C++:

```cpp
#include <iostream>
using namespace std;
class A {
    private:
        int x;
    public:
        A() { x = 0; }
        void setX(int val) {
          x = val;
          }
        friend class B; // class B is now a friend of A
};

class B {
    public:
        void printX(A &a) {
            cout << "Value of x in class A: " << a.x << endl;
             // because class B is a friend of A, it can access A's private members
        }
};

int main() {
    A a;
    a.setX(10);
    B b;
    b.printX(a); // prints the value of x in class A which is 10
    return 0;
}
```

Output :-

Value of x in class A: 10

In this example, class A has a private member variable "x" and a public function "setX" to set the value of x. Class B is declared as a friend of class A and has a public function "printX" which takes an object of class A as an argument and prints the value of x. The main function creates an object of class A, sets the value of x to 10 and creates an object of class B, then it calls the printX function of class B which prints the value of x.

Advantages of Friend Function:

- **Access to private and protected members**: Friend functions have access to the private and protected members of a class, which allows them to perform operations that would not be possible with public member functions.
- **Code Reusability:** Friend functions can be reused in multiple classes, which can help to reduce code duplication and improve maintainability.
- **Encapsulation:** Friend functions can be used to implement complex relationships between classes without breaking encapsulation.
- **Implementation Hiding:** Friend functions can be used to hide the implementation details of a class from other classes.
- **Improved performance:** Friend functions can be implemented more efficiently than member functions, as they can bypass the overhead of the this pointer.

- **Overloading operator:** Friend functions can be used to overload operators that cannot be overloaded by member functions.

Disadvantages of Friend Function:

- **Tight Coupling:** Friend functions can lead to tight coupling between classes, which can make the code harder to understand and maintain.
- **Inefficient code:** Using friend functions can make the code less efficient, as they can bypass the normal member function call mechanism.
- **Security Issues:** Friend functions can be used to access and modify private and protected members of a class, which can lead to security issues if not used properly.
- **Maintainability:** Friend functions can make the code harder to understand and maintain, as the relationships between classes can become complex and hard to follow.
- **Non-member function:** A friend function is not a member of the class, so it can't be virtual or be overridden, which can limit the flexibility of the class design.

• Exception Handling :-

Exception handling in C++ is a mechanism that allows you to handle runtime errors and unexpected situations in your code. It allows you to separate the error-handling code from the main logic of your program, making the code easier to read and understand.

Exception Handling Keywords:

In C++, we use 3 keywords to perform exception handling:

1. **try** : Represents a block of code that can throw an exception.
2. **catch** : Represents a block of code that is executed when a particular exception is thrown.
3. **throw** : Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

The basic syntax for exception handling in C++ is as follows:

```
try {
    // code that might throw an exception
} catch (exception_type variable_name) {
    // error-handling code
}
```

- The **try** block encloses the code that might throw an exception.
- The **catch** block is used to handle the exception that is thrown. It takes an exception object of a specified type as an argument.
- The **exception_type** is the type of exception that you want to catch. This can be a standard exception class, such as std::exception, or a custom exception class that you define.
- The **variable_name** is an optional variable that you can use to access the exception object and get more information about the error that occurred.

Example :-

1. The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

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

int divide(int a, int b) {
    if (b == 0) {
        throw runtime_error("Cannot divide by zero");
    }
    return a / b;
}

int main() {
    cout << "Before try block" << endl;
    try {
        int x = divide(10, 0);
        cout << "The result is: " << x << endl;
    }
    catch (const runtime_error& e) {
        cout << "Error: " << e.what() << endl;
    }
    cout << "After catch block" << endl;
    return 0;
}
```

Output :-

Before try block
Error: Cannot divide by zero After catch block

In this example, the **divide** function takes two integers as input and returns the division of the first number by the second one. If the second number is zero, it throws an exception of type **runtime_error** with an error message "Cannot divide by zero". The **try** block in the **main** function calls the **divide** function. If the exception is thrown, the program jumps to the catch block, which catches the exception and prints the error message. Finally, the program continues execution and prints "After catch block"

2. There is a special catch block called the ‘catch all’ block, written as catch(...), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(...) block will be executed.

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

void throw_exception() {
    throw 42;
}

int main() {
    try {
        throw_exception();
    } catch (double e) {
        cout << "Caught double exception: " << e << endl;
    } catch (const char* e) {
        cout << "Caught const char* exception: " << e << endl;
    } catch (...) {
        cout << "Caught an unknown exception" << endl;
    }
    return 0;
}
```

Output :-

Caught an unknown exception

3. Implicit type conversion doesn't happen for primitive types. For example, in the following program, 'a' is not implicitly converted to int.

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

int main()
{
    try  {
        throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output :-

Default Exception

part :- 2 soon.....