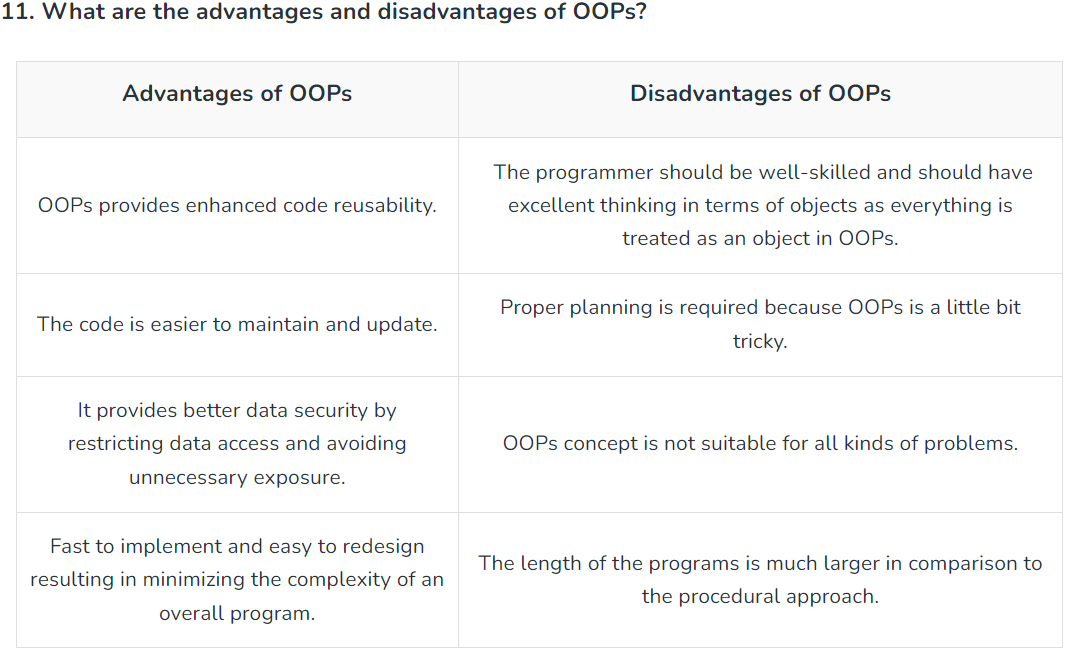
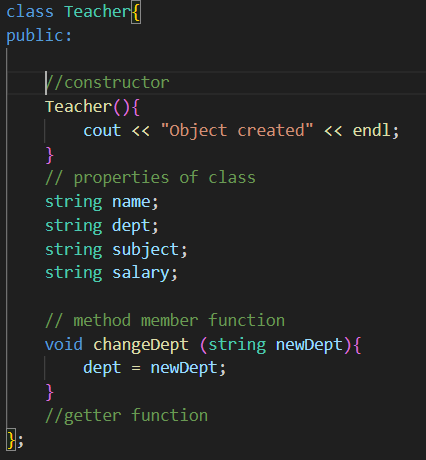
OBJECT ORIENTED ROGRAMMING

* **Object-Oriented Programming**, or OOPs, is a programming paradigm that implements the concept of objects in the program. It aims to provide an easier solution to real-world problems by implementing real-world entities such as inheritance, abstraction, polymorphism, etc. in programming.



* **Class** is a user-deﬁned data type that contain data member and member function. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

C++ Syntax (for class) :

* 1. Class name should be in CamelCase (common convention).
  2. Constructor name should be same as class name.
  3. Object are entities in the real world and class are the blue print for this entities.
* **Object:** An **object** is an instance of a class. Data members and methods of a class cannot be used directly. We need to create an object (or instance) of the class to use them. In simple terms, they are the actual world entities that have a state and behavior.

C++ Syntax (for object):

student s = **new** student();

**Note** : When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created **without** a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.



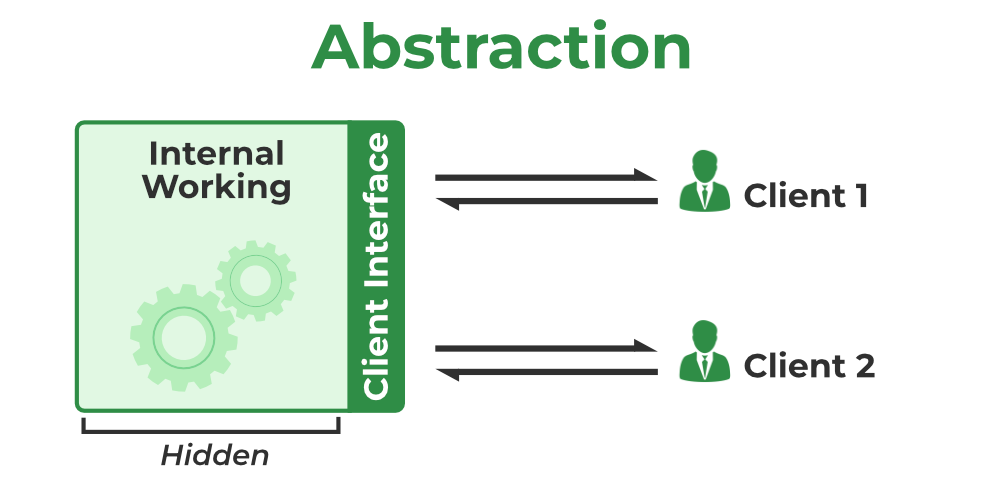
## Encapsulation

Encapsulation is the process of combining data and functions into a single unit called class such that the sensitive data is hidden from the users. It is implemented as the processes mentioned below:

1. **Data hiding:** A language feature to restrict access to members of an object. For example, private and protected members in C++.
2. **Bundling of data and methods together:** Data and methods that operate on that data are bundled together. For example, the data members and member methods that operate on them are wrapped into a single unit known as a class.

## Abstraction

Abstraction is similar to data encapsulation and is very important in OOP. It means showing only the necessary information and hiding the other irrelevant information from the user. Abstraction is implemented using classes and interfaces.

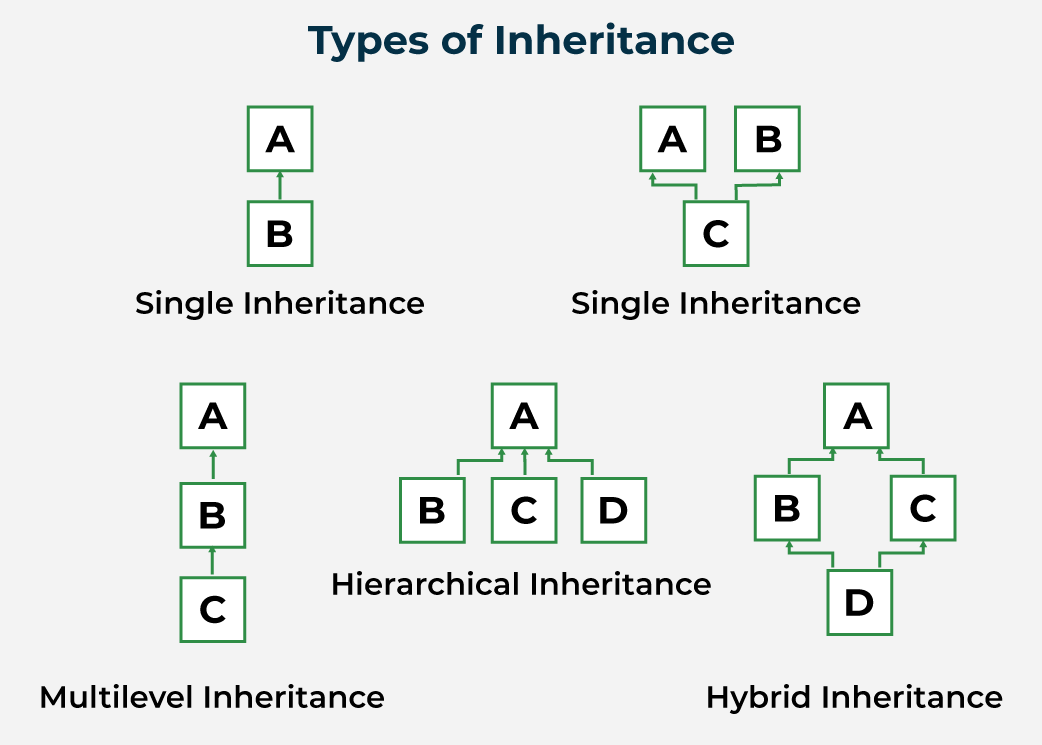


## Inheritance

The idea of inheritance is simple, a class is derived from another class and uses data and implementation of that other class. The class which is derived is called child or derived or subclass and the class from which the child class is derived is called parent or base or superclass.

The main purpose of Inheritance is to increase code reusability. It is also used to achieve Runtime Polymorphism.

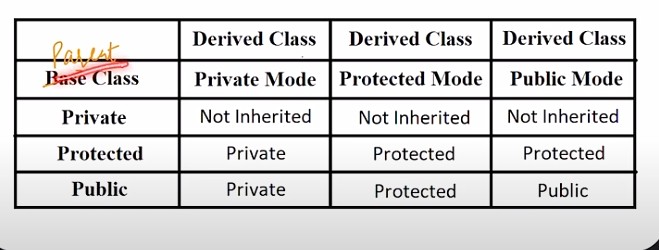
**Types of Inheritance:**



* + 1. **Single inheritance :** When one class inherits another class, it is known as

single level inheritance

* + 1. **Multiple inheritance:** Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.
    2. **Hierarchical inheritance:** Hierarchical inheritance is deﬁned as the process of deriving more than one class from a base class.
    3. **Multilevel inheritance:** Multilevel inheritance is a process of deriving a class from another derived class.
    4. **Hybrid inheritance:** Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

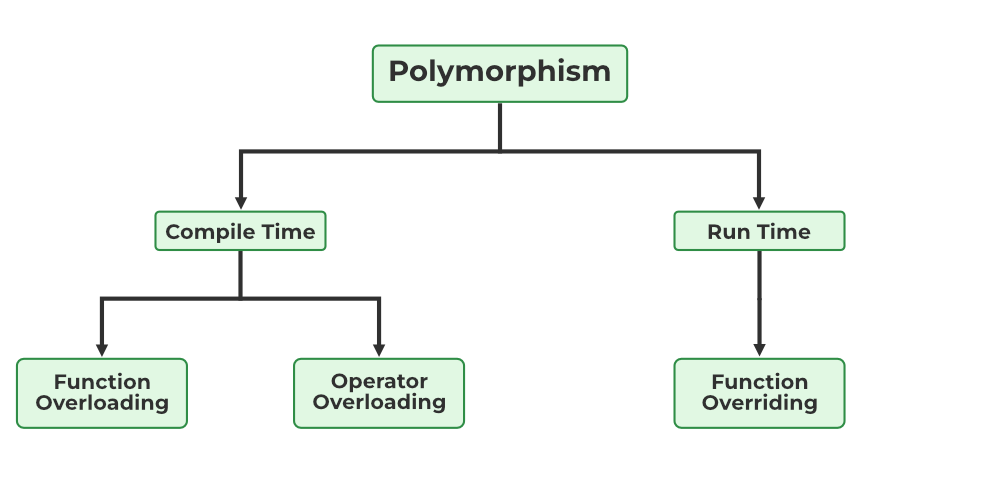


## Polymorphism

The word “**Polymorphism**” means having many forms. It is the property of some code to behave differently for different contexts. For example, in C++ language, we can define multiple functions having the same name but different working depending on the context.

Polymorphism can be classified into two types based on the time when the call to the object or function is resolved. They are as follows:

* Compile Time Polymorphism
* Runtime Polymorphism



**A) Compile-Time Polymorphism**

Compile time polymorphism, also known as static polymorphism or early binding is the type of polymorphism where the binding of the call to its code is done at the compile time. Method overloading or operator overloading are examples of compile-time polymorphism.

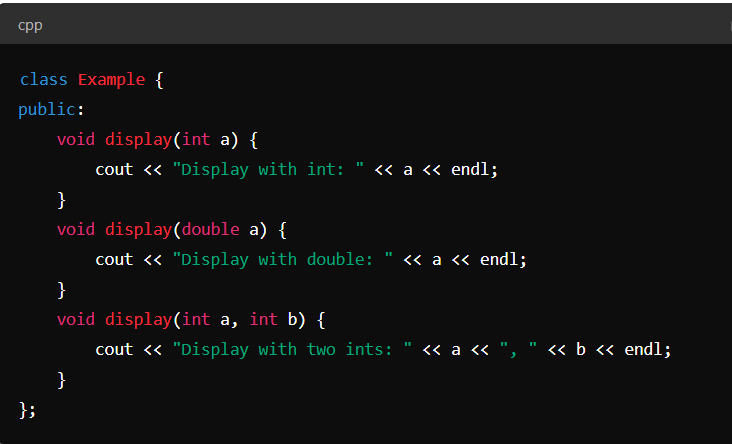
**B) Runtime Polymorphism**

Also known as dynamic polymorphism or late binding, runtime polymorphism is the type of polymorphism where the actual implementation of the function is determined during the runtime or execution. Method overriding is an example of this method.

### Difference Between Method Overloading and Method Overriding

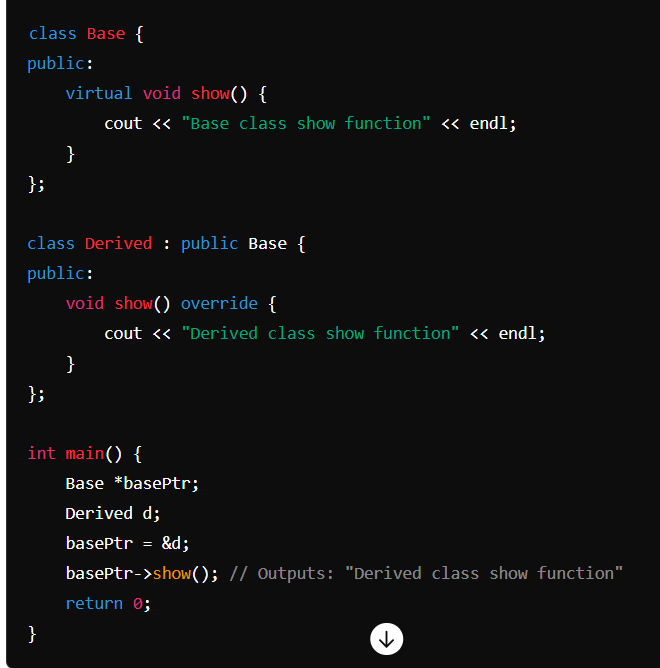
#### Method Overloading:

1. **Definition**: Method overloading allows multiple methods in the same class to have the same name but different parameters (different type, number, or both).
2. **Binding Time**: Compile-time (static binding).
3. **Purpose**: It is used to increase the readability of the program and to perform different tasks with the same method name by changing the method signature.
4. **Parameters**: Methods must have different parameter lists (different type, number, or both).
5. **Inheritance**: Overloading does not require inheritance. Methods can be overloaded within the same class.
6. **Return Type**: Return type can be different or the same, but it alone cannot distinguish overloaded methods.
7. **Example**:



#### Method Overriding:

1. **Definition**: Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
2. **Binding Time**: Runtime (dynamic binding).
3. **Purpose**: It is used to achieve runtime polymorphism and to implement specific behavior for subclasses.
4. **Parameters**: Method signature (name and parameters) must be exactly the same in both the superclass and subclass.
5. **Inheritance**: Overriding requires inheritance. It occurs in a subclass.
6. **Return Type**: Return type must be the same or covariant (in case of reference types in some languages like Java).



### Summary:

* **Method Overloading** deals with multiple methods in the same class with the same name but different parameters, and it resolves at compile time.
* **Method Overriding** deals with redefining a method in a derived class that is already defined in its base class, and it resolves at runtime.

**Access Specifiers:**

Access specifiers are special types of keywords that are used to specify or control the accessibility of entities like classes, methods, and so on. **Private**, **Public**, and **Protected** are examples of access specifiers or access modifiers.  
The key components of OOPs, encapsulation and data hiding, are largely achieved because of these access specifiers.

* **Constructor** : Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure also constructor don't have any return type.

There can be **two types** of constructors in C++.

1. Default constructor : A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.
2. Parameterized constructor : Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.
3. Copy Constructor : A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. It is of two types - default copy constructor and user deﬁned copy constructor.

* **Destructor** : A destructor works opposite to constructor; it destructs the objects of classes. It can be deﬁned **only once** in a class. Like constructors, it is invoked automatically. A destructor is deﬁned like a constructor. It must have the same name as class, preﬁxed with a **tilde sign (~)**.

A a;

A b;

**return** 0;

}

/\*

Output: Constructor in use

Constructor in use Destructor in use Destructor in use

\*/

* **‘this’ Pointer** : **this** is a keyword that refers to the **current instance of the class**. There can be 3 main uses of ‘this’ keyword:

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

C++ Syntax :

## struct node{

**int** data; node \*next;

node(**int** x){

**this**->data = x; **this**->next = **NULL**;

}

}

* **Friend Function** : Friend function acts as a friend of the class. **It can access the private and protected members of the class.** The friend function is not

a member of the class, but it must be listed in the class deﬁnition. The non-member function cannot access the private data of the class.

Sometimes, it is necessary for the non-member function to access the data. **The friend function is a non-member function and has the ability to access the private data of the class**.

Note :

1. A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
2. Friend function uses objects as arguments.

## Example IMP :

#include <bits/stdc++.h>

## using namespace std;

**class A**{

**int** a = 2; **int** b = 4; **public**:

// friend function **friend int mul**(A k){ **return** (k.a \* k.b);

}

};

**int main**(){

A obj;

**int** res = mul(obj); cout << res << endl; **return** 0;

}

// Output : 8

* **Aggregation :** It is a process in which one class deﬁnes another class as

any entity reference. **It is another way to reuse the class**. It is a form of association that represents the HAS-A relationship.

* **Virtual Function IMP**: A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.

## A virtual function is a member function which is present in the base class and redeﬁned by the derived class.

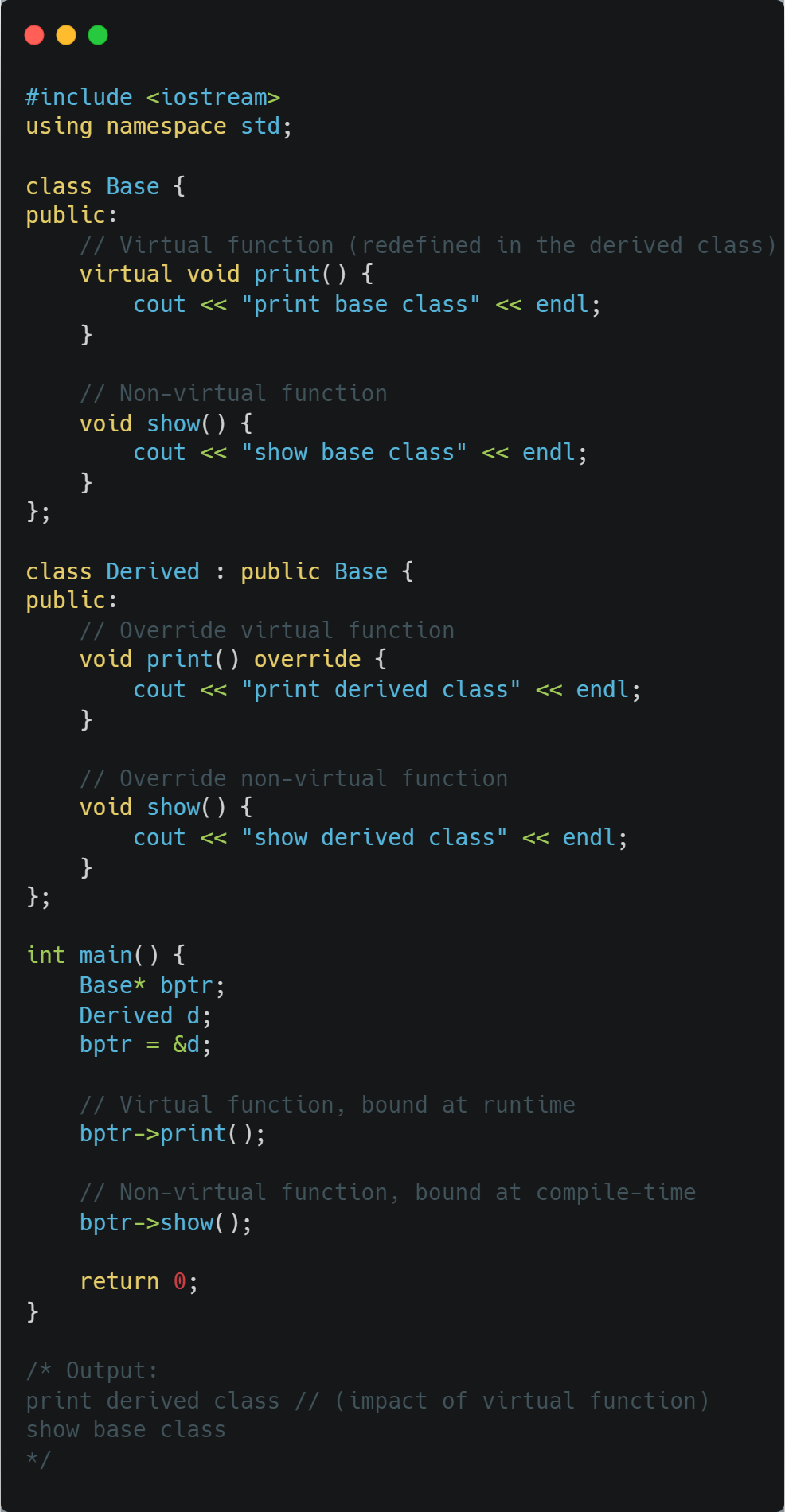
1. When we use the same function name in both base and derived class, **the function in base class is declared with a keyword virtual.**
2. When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class

## pointer. Thus, by making the base class pointer to point to different objects, we can execute different versions of the virtual functions.

**Key Points** :

1. Virtual functions cannot be static.
2. A class may have a virtual destructor but it cannot have a virtual constructor.

## C++ Example :



## Pure Virtual Function :

1. A pure virtual function is not used for performing any task. It only serves as a placeholder.
2. A pure virtual function is a function declared in the base class that has no deﬁnition relative to the base class.
3. A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as **abstract base classes**.
4. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.



**Output:**

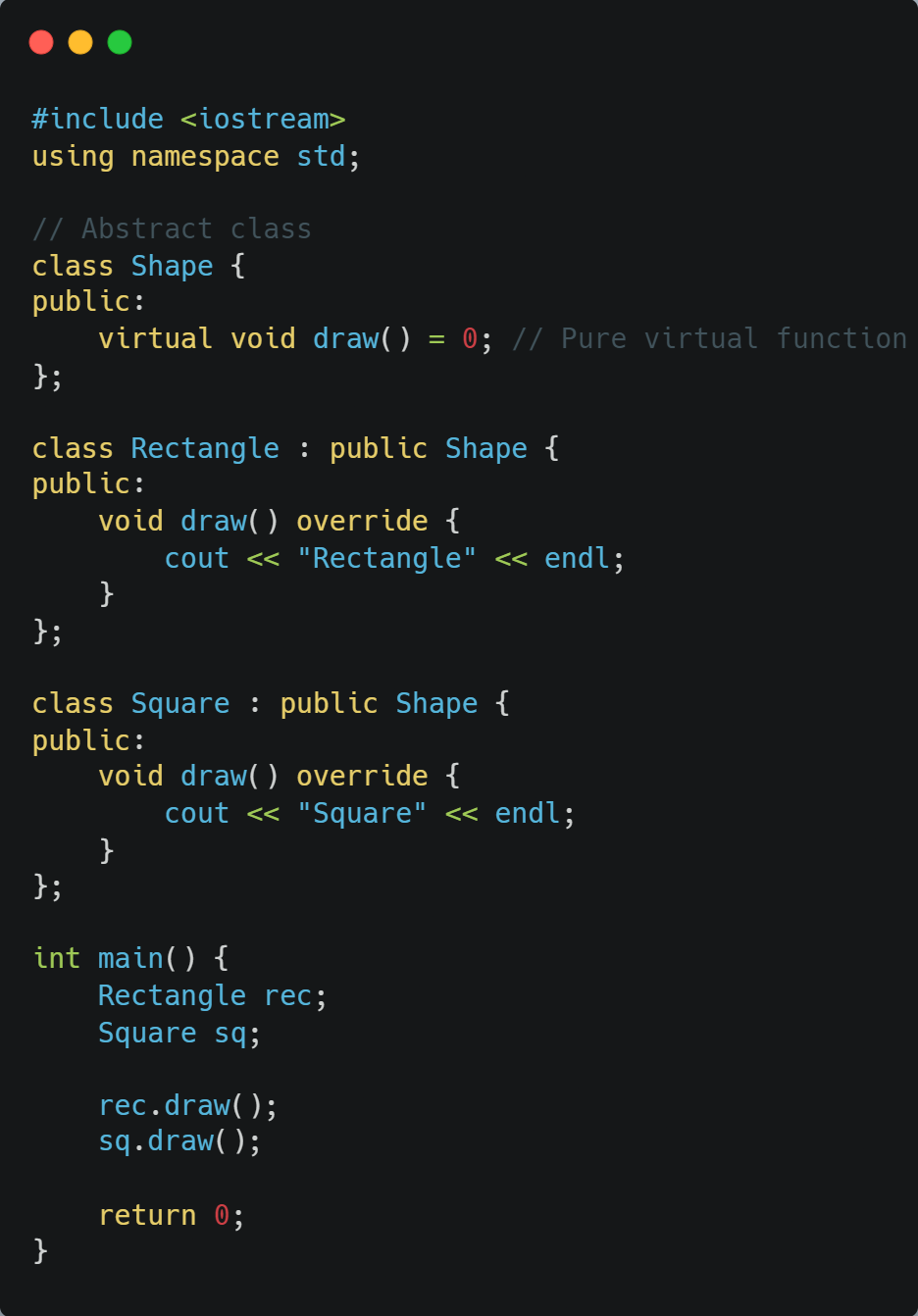
**Now you can see me!**

1. **Pure Virtual Function**: virtual void show() = 0; makes Base an abstract class, which means it cannot be instantiated.
2. **Override in Derived Class**: The Derived class provides the implementation for the pure virtual function show.
3. **Polymorphic Behavior**: A Base class pointer can point to a Derived class object, and calling show() on the pointer will invoke the Derived class's implementation of show().

### Output:

* **Abstract Classes**: In C++ class is made abstract by declaring at least one of its functions as a **pure virtual function**. A pure virtual function is speciﬁed by placing "= 0" in its declaration. **Its implementation must be provided by derived classes.**

## Example :



rec.draw();

sq.draw();

**return** 0;

}

/\* Output :

Rectangle Square

\*/

## Namespaces in C++ :

1. The namespace is a logical division of the code which is designed to stop the naming conﬂict.
2. The namespace deﬁnes the scope where the identiﬁers such as variables, class, functions are declared.
3. **The main purpose of using namespace in C++ is to remove the ambiguity.** Ambiguity occurs when a different task occurs with the same name.
4. For example: if there are two functions with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
5. C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.

## C++ Example :

#include <bits/stdc++.h>

## using namespace std;

// user-deﬁned namespace

**namespace** Add { **int** a = 5, b = 5;

**int add**() {

**return** (a + b);

}

}

**int main**() {

**int** res = Add :: add(); // accessing the function inside namespace cout << res;

}

// output : 10

* **Access Speciﬁers IMP :** The access speciﬁers are used to deﬁne how functions and variables can be accessed outside the class. There are three types of access speciﬁers:

1. **Private**: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared by default both properties and member functions are **Private**.
2. **Public**: Functions and variables declared under public can be accessed from anywhere.
3. **Protected**: Functions and variables declared as protected cannot be accessed outside the class except a child class. This speciﬁer is generally used in inheritance.

## Key Notes

* **Delete** is used to release a unit of memory, **delete[]** is used to release an array.
* **Virtual inheritance** facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.
* **Function overloading:** Function overloading is deﬁned as we can have more than one version of the same function. The versions of a function will have different signatures meaning that they have a different set of parameters.

**Operator overloading:** Operator overloading is deﬁned as the standard operator can be redeﬁned so that it has a different meaning when applied to the instances of a class.

* **Overloading** is static Binding, whereas Overriding is dynamic Binding. Overloading is nothing but the same method with different arguments, and it may or may not return the same value in the same class itself. **Overriding** is the same method name with the same arguments and return types associated with the class and its child class.

# Shallow Copy

A shallow copy creates a new object and then copies the non-static fields of the current object to the new object. If the field is a reference to an object, the reference is copied, meaning both the original and the copy will point to the same object in memory. This can lead to unexpected behavior if the shared object is modified from either reference.

# Example:

In the Student class example provided, the shallow copy constructor simply copies the pointer cgpa from the original Student object to the new one. Thus, both objects share the same memory address for cgpa.

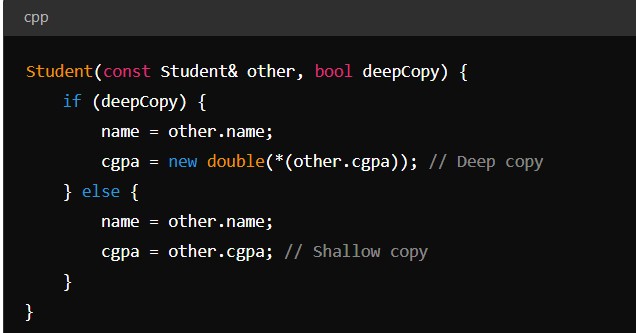
# Key Point:

* 1. Fast and memory efficient.
  2. Risk of unintended side effects due to shared references.

# Deep Copy

A deep copy creates a new object and then copies all fields, creating duplicates of dynamically allocated memory that the fields reference. This means the new object and the original object do not share the same memory addresses for their references, thus modifications to one do not affect the other.

# Example:

In the Student class example, the deep copy constructor allocates new memory and copies the value pointed to by cgpa from the original object.

# Key Point:

1. Ensures complete independence between the original and copied objects.
2. More resource-intensive compared to a shallow copy due to additional memory allocation.

[**https://www.geeksforgeeks.org/oops-interview-questions/**](#_bookmark0)

**### Abstraction:**

# \*\*Using Abstract Classes\*\*

Abstract classes are used to provide a base class from which other classes can be derived. They cannot be instantiated and are meant to be inherited. Abstract classes are typically used to define an interface for derived classes.

Read about it what is use of it .....